Think Like a Vertex, Behave Like a Function! A Functional DSL for Vertex-Centric Big Graph Processing

Kento Emoto
Kyushu Institute of Technology, Japan
emoto@ai.kyutech.ac.jp

Kiminori Matsuzaki
Kochi University of Technology, Japan
matsuzaki.kiminori@kochi-tech.ac.jp

Zhenjiang Hu
National Institute of Informatics, Japan
hu@nii.ac.jp

Akimasa Morihata
The University of Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp

Hideya Iwasaki
The University of Electro-Communications, Japan
iwasaki@cs.uec.ac.jp

Abstract

The vertex-centric programming model, known as "think like a vertex", is being used more and more to support various big graph processing methods through iterative *supersteps* that execute in parallel a user-defined vertex program over each vertex of a graph. However, the imperative and message-passing style of existing systems makes defining a vertex program unintuitive. In this paper, we show that one can benefit more from "Thinking like a vertex" by "Behaving like a function" rather than "Acting like a procedure" with full use of side effects and explicit control of message passing, state, and termination. We propose a functional approach to vertex-centric graph processing in which the computation at every vertex is abstracted as a higher-order function and present Fregel, a new domain-specific language. Fregel has clear functional semantics, supports declarative description of vertex computation, and can be automatically translated into Pregel, an emerging imperative-style distributed graph processing framework, and thereby achieve promising performance. Experimental results for several typical examples show the promise of this functional approach.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; E.1 [*Data Structures*]: Graphs and networks

Keywords Pregel, big graph processing, domain-specific language, skeletal parallel programming

1. Introduction

The rapid growth of large-scale graphs in real life is driving demand for dependable and productive programming models for supporting highly efficient and parallel implementation of big graph processing. The *vertex-centric* programming model (McCune et al. 2015) is one such model. It has been intensively studied and has served as the basis for a number of practically useful distributed graph processing systems, such as Pregel (Malewicz et al. 2010) and GraphLab (Low et al. 2012). In contrast to the randomly accessible, global perspective of data reflected in conventional graph computation models, the vertex-centric computation model reflects a *vertex-local* perspective of computation, encouraging practitioners to "think like a vertex". This vertex-centric approach has demonstrated its advantages in improving locality and scalability for large-scale graph processing.

Pregel (Malewicz et al. 2010), one of the most promising frameworks for big graph processing, uses a vertex-centric computation model based on the bulk synchronous parallel (BSP) model of computation (Valiant 1990). In Pregel, various graph computations are achieved through iterative *superstep*s. Each superstep consists of parallel execution of a user-defined program over vertices distributed among computation nodes, communications between vertices, and global barrier synchronization, which ensures the delivery of communication messages. The user-defined vertex function typically accepts messages sent from incoming edges as input, sends results to other vertices along outgoing edges, and will vote for halt if nothing needs to be done.

As an example, we consider a simple problem of marking all vertices of a graph that are reachable from the source vertex (vertex id = 0). We call it the "all-reachability problem" hereafter. Using the vertex-centric approach, we define the vertex function by using the pseudo code presented in Fig. 1 and iteratively execute it on all vertices. If all vertices are initially marked `false`, the vertex program in Fig. 1 accepts a vertex and its received messages as input and marks the vertex if it is a neighbor of a vertex marked reachable from the source vertex. If the program is at the first superstep (Line 2), the source node is marked `true`, and this information is sent to the neighboring vertices. Otherwise, the vertex checks whether there is any message containing `true` but the vertex has not been marked yet. If this is the case, the vertex marks itself (Line 9) and notifies the neighboring vertices (Line 10).

During this execution, each vertex is either *active* or *inactive*. Initially all vertices are active. A vertex becomes inactive by calling the `voteToHalt` function (Lines 5 and 12). In each superstep, only active vertices execute their calculations and send messages if necessary. An inactive vertex becomes active again by being sent a message from another vertex. The entire processing for a graph terminates when all vertices become inactive and there remain no unreceived messages. Figure 2 demonstrates how three supersteps are used to mark all reachable vertices for an input graph with five vertices.

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951938

```
1  vertex.compute(v, messages) {
2   if(superstep == 0) {
3     v.rch = v.vid == 0;
4     if(v.rch) sendToNeighbors(v.rch);
5     else voteToHalt();
6   } else {
7     newrch = v.rch || or(messages);
8     if(newrch != v.rch) {
9       v.rch = newrch;
10      sendToNeighbors(newrch);
11    }
12    voteToHalt();
13  }
14 }
```

Figure 1. Pregel-like code for all-reachability problem.

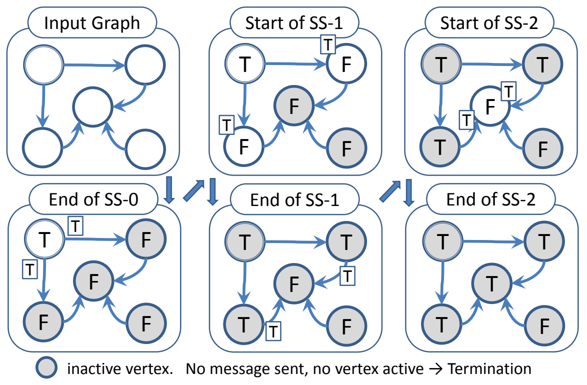

Figure 2. Example of marking all reachable vertices using three supersteps (SSs); with the left being SS0, the middle being SS1, and the right being SS2.

Despite the power of the vertex-centric approach, the imperative and message-passing style of existing systems makes defining the vertex program unintuitive. We can see several problems from the program in Fig. 1.

- **Full use of side effects**. `vertex.compute` is a procedure rather than a function. It is imperative in the sense that it computes and passes important results through *side effects*, updating the local vertex value and sending information through implicit channels.
- **Explicit message-passing**. The behavior of `vertex.compute` is difficult to understand because it requires inferring backwardly the meaning of the input messages from how and what messages are sent (in the previous superstep) as defined in the body. This becomes more difficult as the vertex program gets bigger and more complex.

To see more problems, suppose that we want to mark the reachable vertices and stop when we have a sufficient number of them (say 100). For simplicity, we assume that there are more than 100 reachable vertices in the target graph. We call this the "100-reachability problem". At each superstep, we need to count the number of currently reachable vertices to determine whether we should go further or halt. To enable acquiring such global information, Pregel supports a mechanism called *aggregation*, which collects data from all active vertices and aggregates them by using a specified operation such as sum or max. Every vertex can use the aggregation result in the next superstep. By using aggregation to count the number of vertices that are marked `true` (Line 9), we can solve the 100-reachability problem by using the vertex program in Fig. 3. Note that aggregation should be done before the check for halting (Line 12). This order is guaranteed by using the odd supersteps to compute aggregation and the even supersteps to do checking (Line 7). The value of `newrch` is set in an odd superstep (Line 8) and read in the next even superstep (Line 17). Thus, we have to change `newrch` from a local variable in Fig. 1 to a member variable of a vertex in Fig. 3. This example shows an additional problem.

```
1  vertex.compute(v, messages) {
2   if(superstep == 0) {
3     v.rch = v.vid == 0;
4     if(v.rch) sendToNeighbors(v.rch);
5     else voteToHalt();
6   } else {
7     if(superstep % 2 == 1) {
8       v.newrch = v.rch || or(messages)
9       aggregate("cnt", v.newrch ? 1 : 0);
10    } else {
11      cnt = read_agg("cnt");
12      if(cnt > 100) {
13        voteToHalt();
14        return;
15      }
16      if(v.newrch != v.rch) {
17        v.rch = v.newrch;
18        sendToNeighbors(v.newrch);
19      }
20    }
21  }
22 }
```

Figure 3. Pregel-like code for 100-reachability problem.

- **Explicit state control**. Because aggregation is used to acquire global information, programmers must explicitly assign states to supersteps so that different supersteps behave differently, resulting in dependency consistency between the local states and the global information.

In the program in Fig. 3, `voteToHalt()` is carefully used. On the one hand, all vertices marked `true` should be active because they should take part in the aggregation. On the other hand, they should finally vote to halt for termination. As a consequence, termination of the whole system is far from obvious. For instance, if we simply swapped the "if" statement in Lines 12–15 and the "if" statement in Lines 16–19, we would have a vertex program that continues marking even after finding *all* reachable vertices. This reveals the following problem.

- **Explicit termination control**. The complicated uses of the termination control statement `voteToHalt()` in the vertex program for explicitly stopping the vertex would make it hard to write a program that terminates appropriately.

Why do we have these problems? How can we avoid these problems? The explicit message-passing style could be avoided if we changed the thinking of vertex computation from passive action over the pushed messages to active computation on each vertex with peeking in neighboring vertices for information necessary for computation. This would change the vertex program from a procedure to a function, which is our solution to the problems.

Given this observation, here we show that one can benefit more from "Thinking like a vertex" by "Behaving like a function" rather than "Acting like a procedure" with full use of side effects and explicit control of message passing, state, and termination. We present a new functional model for vertex-centric graph processing and describe the design and implementation of Fregel, a functional domain-specific language (DSL), which not only supports declarative description of vertex computation but can be automatically translated into code runnable on existing systems.

Our technical contributions can be summarized as follows.

Peek-based vertex-centric functional model. We abstract and formalize the Pregel computation as a higher-order function that captures the higher-level behavior of Pregel computation by using a recursive execution corresponding to dynamic programming on a graph (Section 2). In contrast to the traditional vertex-centric computation model, which pushes (sends) information from a vertex to other vertices so that the vertex computation can be done in the next superstep, our model is peek-based in the sense that a vertex peeks (receives) necessary information from other vertices to enable computation on the vertex at the current superstep.

Functional DSL for programming vertex-centric graph processing. We propose Fregel, a functional DSL for declarative-style programming on big graphs that is based on the peeking-style vertex-centric model. It abstracts aggregation and communication by using comprehension with a specific generator. Fregel also encourages concise, compositional-style programming on big graphs by providing four higher-order functions on graphs. It allows arbitrary nesting of expressions that include operations over graphs, as opposed to having only one top-level graph-oriented computation in Pregel. Fregel is purely functional without any side effects. This functional nature enables various transformations during the compilation of Fregel programs. As Fregel is a subset of Haskell, it is possible to use Haskell tools to test and debug Fregel programs.

Implementation with promising performance. We show that a Fregel program can be compiled into a single Giraph[1] program with reasonable and promising performance. The compilation process is two-fold. First, given a Fregel program, normalization flattens the program's multiple uses of the graph higher-order functions into a single use of one general higher-order function. Then, code generation produces a Giraph program from the normalized Fregel program by simply mapping the general higher-order function to a single run of Giraph computation. This flattening is important for reasonable performance; a naive compilation that mapped each graph higher-order function to a single Giraph computation without flattening would suffer heavy start-up costs of multiple runs of Giraph computation. This compilation process frees programmers from the problematic programming burden, e.g., splitting supersteps into two modes (sending/referring data) for aggregation. We have implemented and tested our method with several nontrivial examples. The experimental results show the promise of our approach.

The organization of the rest of the paper is as follows. We start in Section 2 by presenting our peek-based functional model for vertex-centric computation and then abstract it as a higher order function. Next, we define Fregel, a functional DSL for declarative-style programming on big graphs in Section 3, the normalization procedure in Section 4, and the automatic compilation in Section 5. Evaluation results are reported in Section 6. Related work is discussed in Section 7. Section 8 concludes with a summary of the key points and a look at future work.

To avoid confusion, we use `this font` to denote code fragments in Haskell and *`this font`* to denote those in Fregel except for some reserved words.

2. Functional Model for Pregel

We modeled the vertex-centric computation in Pregel as a higher-order function and designed Fregel, a functional DSL, on the basis of this model. In this section, we introduce our model by using the notation of Haskell.

[1] an open source implementation of Pregel: http://giraph.apache.org/

In this paper, we impose the following restrictions on Pregel.

- Computation on a vertex does not change the shape of the graph.
- Each message is exchanged between two adjacent vertices directly connected by a directed edge.

A model with the latter restriction is called the GAS (gather-apply-scatter) model and is regarded as realistic (Bae and Howe 2015; Gonzalez et al. 2012; Sengupta et al. 2015). Under these restrictions, it impossible to represent algorithms that apply the pointer jumping technique to the graph. Nevertheless, we impose these restrictions because we want to establish the foundation for an appropriate model for Pregel.

2.1 Definition of Datatypes

First, we define the datatypes needed for our functional model. Let `Graph a b` be the type of graph for which the vertices have the type `Vertex a b` and the *incoming* edges have the type `Edge a b`. `Graph a b` is a list of vertices, each of which has the type `Vertex a b`. A vertex of type `Vertex a b` has a unique vertex identifier, a value of type `a`, and a list of incoming edges with type `Edge a b`. An edge of type `Edge a b` is a pair of the edge value of type `b` and the adjacent vertex connected by this edge.

```
data Vertex a b =
  Vertex { vid :: Int, val :: a, is :: [Edge a b] }
type Edge a b = (b, Vertex a b)
type Graph a b = [Vertex a b]
```

As an example, the lower-right graph (End of SS-2) in Fig. 2 can be defined by the following cyclic data structure, where `v0`, `v1`, `v2`, `v3`, and `v4` are the upper-left, upper-right, lower-left, middle, and lower-right vertices, respectively. We assume that all edges have the value 1.

```
g :: Graph Bool Int
g = let v0 = Vertex 0 True []
        v1 = Vertex 1 True [(1,v0)]
        v2 = Vertex 2 True [(1,v0)]
        v3 = Vertex 3 True [(1,v1),(1,v2),(1,v4)]
        v4 = Vertex 4 False []
    in [v0, v1, v2, v3, v4]
```

2.2 Description of our Model

In vertex-centric parallel computation, each vertex periodically and synchronously performs the following processing steps, which we call a *logical superstep*, or *LSS* for short, in the rest of this paper.

1. Each vertex receives the data computed in the previous LSS from the adjacent vertices connected by incoming edges.
2. In accordance with the problem to be solved, the vertex does its computation by using the received data and the data it computed in the previous LSS. If necessary, the vertex acquires global information that is needed by using aggregation during its computation.
3. The vertex sends the results of the computation to all adjacent vertices along its outgoing edges. Every adjacent vertex receives the data in the next LSS.

An LSS is "logical" because it might contain aggregation and thus might be realized by more than one Pregel superstep.

We represent an LSS as a function and call it *LSS function*. As explained later, an LSS function does not explicitly describe sending and receiving data; instead it uses recursive calls on itself.

The arguments of an LSS function are an integer value (the *clock*), which represents the number of iterations of the LSS function, and the vertex on which the LSS function is repeatedly performed. Thus, the type of an LSS function is `Int -> Vertex a b -> r`, where `r` is the type of the result. Let `lss` be an LSS function. Then, `lss 0 v` represents the initial value of the computation on vertex `v`. In addition, `lss t v` for $t > 0$ performs computation on vertex `v` at clock `t` by using its own result at clock `t-1`, i.e., `lss (t-1) v`, and every result at clock `t-1` of an adjacent vertex connected by an incoming edge, i.e., `lss (t-1) u`, where `u` is the adjacent vertex. Such an LSS function can be characterized by two functions. One is an *initialization* function, which defines the behavior for $t = 0$, and the other is a *step* function, which defines the behavior for $t > 0$. On the basis of these two functions, a general form of LSS function, `lssGen`, can be defined as follows.

```
lssGen :: (Vertex a b -> r) ->
          (r -> [(b,r)] -> Vertex a b -> r) ->
          Int -> Vertex a b -> r
lssGen f0 ft 0 v = f0 v
lssGen f0 ft t v =
  ft (lssGen f0 ft (t-1) v)
     [(e, lssGen f0 ft (t-1) u) | (e,u) <- is v]
     v
```

An LSS function for a specific problem is defined by giving appropriate initialization and step functions, say `init` and `step`, as arguments to `lssGen`, i.e., `lss = lssGen init step`.

Let g = [v_0, v_1, v_2, ...] be the target graph of the Pregel computation. The list of computation results of LSS function `lss` on all vertices in the graph at clock `t` is [lss t v_0, lss t v_1, lss t v_2, ...]. In addition, let makeGraph g [r_0, r_1, r_2, ...] return a graph with the same shape as g for which the i-th vertex has the value r_i and the edges have the same values as those in g. Then, the infinite list of graphs

[makeGraph g [lss 0 v_0, lss 0 v_1, lss 0 v_2, ...],
 makeGraph g [lss 1 v_0, lss 1 v_1, lss 1 v_2, ...],
 makeGraph g [lss 2 v_0, lss 2 v_1, lss 2 v_2, ...],
 ...],

for which the t-th element corresponds to a graph constructed from the results of `lss` on all vertices at clock t, represents infinite iterations of LSS function `lss`. The infinite list can be produced by using the following higher-order function `pregelIter`, which takes as its arguments initialization and step functions for `lss` and a target graph (a list of vertices).

```
pregelIter :: (Vertex a b -> r) ->
              (r -> [(b,r)] -> Vertex a b -> r) ->
              Graph a b -> [Graph r b]
pregelIter f0 ft g =
  map (\t -> makeGraph g (map (lssGen f0 ft t) g))
      [0..]
```

Though `pregelIter` produces an infinite list, we want to terminate its computation at an appropriate point and take the graph at this point as the final result. A typical termination point is when the computation falls into a steady state, after which the infinite list never changes. With this approach, letting `fixedValue :: (Eq r, Eq b) => [Graph r b] -> Graph r b` return the graph of the steady state of a given infinite list, we can obtain the desired result by using `fixedValue (pregelIter init step vs)`. Another potential termination condition is that the results of `lss` satisfy some condition. With this approach, we can use the higher order function `untilValue :: (Graph r b -> Bool) -> [Graph r b] -> Graph r b`, which takes a predicate function specifying the condition and returns the first graph that satisfies this predicate from a given infinite list.

Finally, we define `pregelModel` as the composition of a termination function and `pregelIter`.

```
reInit :: Vertex a b -> Bool
reInit v = vid v == 0

reStep :: Bool -> [(b,Bool)] -> Vertex a b -> Bool
reStep p eqs v = p || or [ q | (e,q) <- eqs ]

reAllPregelModel :: Graph a b -> Graph Bool b
reAllPregelModel =
  pregelModel reInit reStep fixedValue

re100PregelModel :: Graph a b -> Graph Bool b
re100PregelModel =
  pregelModel reInit reStep
    (untilValue ((> 100) . numTrueVertices))
  where numTrueVertices vs = length (filter val vs)
```

Figure 4. Formulation of reachability problems in our model.

```
pregelModel :: (Vertex a b -> r) ->
               (r -> [(b,r)] -> Vertex a b -> r) ->
               ([Graph r b] -> Graph r b) ->
               Graph a b -> Graph r b
pregelModel f0 ft term = term . pregelIter f0 ft
```

We regard the function `pregelModel` as representing Pregel's computation.

2.3 Simple Example

On the basis of the above model of Pregel, the reachability problems can be formulated as shown in Fig. 4, where `reAllPregelModel` is for the all-reachability problem and `re100PregelModel` is for the 100-reachability problem. We assume that `numTrueVertices` returns the number of vertices with a value of `True` for the target graph. The only difference between these two solutions is the termination condition: the all-reachability problem formulation uses `fixedValue` while the 100-reachability problem one uses `untilValue`. Note that the LSS function characterized by `reInit` and `reStep` has no description for the aggregation that appears in the original Pregel code (Fig. 3).

The definitions in Fig. 4 are Haksell functions and thus can be executed in Haskell. However, executing `reAllPregelModel` or `re100PregelModel` is not efficient because there are many duplicate calls of the LSS function for the same arguments. Thus, in our model, we assume a *memorization* mechanism that prevents such duplicate executions.

2.4 Features of our Model

An LSS function defined in terms of `lssGen` has the form of *structural recursion* on the basis of the structure of the input graph. Although a graph has a cyclic structure, a recursive call of an LSS function does not cause an infinite recursion because a recursive call always uses the prior clock, i.e., `t-1`. In addition, assuming a memorization mechanism, as mentioned above, allows us to consider that such a clock-decreasing recursive execution of an LSS function corresponds to performing *dynamic programming* on a graph.

In an LSS function, there is no explicit description of sending or receiving data between adjacent vertices. The original Pregel views data communication as *explicit-poking style* communication, in which a vertex sends data to an adjacent vertex along its outgoing edge. In contrast, our model views data communication as *implicit-peeking style* communication, in which a vertex peeks at data of an adjacent vertex connected by an incoming edge via an argument of the step function. Figure 5 illustrates the difference between these two view.

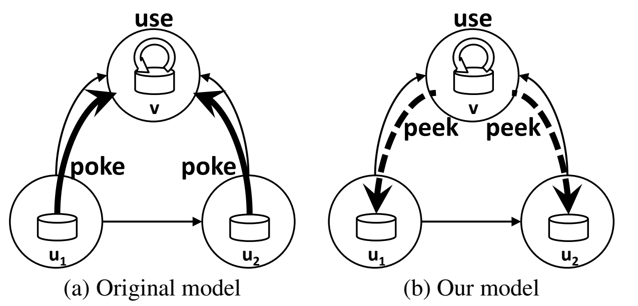

Figure 5. Views of communication. Data on u_1 and u_2 are being used in computation on v. In the original model, the initiators of the communication (u_1 and u_2) are different from the consumer (v) while in our model, they are the same (v peeks at and uses the data).

In summary, our model solves the problems described in the Introduction.

- For the first problem (Full use of side effects), our model is purely *functional*; computation that is periodically and synchronously performed at every vertex is defined as an LSS function without any side effects that have the form of a structural recursion on the graph structure. The recursive execution of such an LSS function is regarded as dynamic programming on the graph on the basis of memorization.
- For the second problem (Explicit message passing), an LSS function has no explicit description of sending or receiving data between adjacent vertices. Instead, it uses recursive calls of the LSS function for adjacent vertices, which can be regarded as an implicit-peeking style of communication.
- For the third problem (Explicit state control), an LSS function describes a processing sequence that is unwillingly divided into small supersteps due to the BSP model's barrier synchronization.
- For the fourth problem (Explicit termination control), the entire computation for a graph is represented as an infinite list of resultant graphs in ascending clock time order. The LSS function has no description for the termination of the computation. Instead, termination is described by a function that appropriately chooses the desired result from an infinite list.

3. Fregel: A Functional DSL for Big Graphs

In this section, we introduce *Fregel*, a functional DSL for declarative style programming on big graphs. Fregel's computation is based on `pregelModel` discussed in Section 2.

3.1 Main Features of Fregel

Fregel captures data access, data aggregation, and data communication in a functional manner and is equipped with four higher-order functions that support concise ways of writing various graph computations in a compositional manner. The main features of Fregel are summarized as follows.

First, Fregel abstracts access to vertex data by using *table*s indexed by vertices. The ***prev*** table is used to access vertex data (i.e., results of recursive calls of LSS function) at the previous clock $t - 1$. The ***curr*** table is used in the normalization of a Fregel program and is described in Sect. 4. These tables explicitly implement the memorization mentioned in Sect. 2. An index given to a table is not the identifier of a vertex or the position of a vertex in a list of incoming edges but is a vertex itself. This enables the programmer to write in a more "direct" style for data accesses.

prog	$:=$	$f\ g\ =\ e$
e	$:=$	$\mathbf{let}\ \mathit{decl}_1\ \cdots\ \mathit{decl}_n\ \mathbf{in}\ e$
	$\mid$	$\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
	$\mid$	$f\ e_1\ \cdots\ e_n$
	$\mid$	$\mathit{comb}\ [\ e\ \mid\ \mathit{gen},\ e_1,\ \ldots,\ e_n\ ]$
	$\mid$	$\mathit{table}\ v\ .\hat{}\ \mathit{fld}_1\ .\hat{}\ \cdots\ .\hat{}\ \mathit{fld}_n$
	$\mid$	***fregel*** $f_1\ f_2\ \mathit{tc}\ g$
	$\mid$	***gmap*** $f\ g$
	$\mid$	***gzip*** $g_1\ g_2$
	$\mid$	***giter*** $f_1\ f_2\ \mathit{tc}\ g$
decl	$:=$	$f\ x_1\ \cdots\ x_n\ =\ e$
	$\mid$	$f\ v$ ***prev curr*** $=\ e$
gen	$:=$	$u \leftarrow g$ $\mid$ $(\mathit{ed}, u) \leftarrow$ ***is*** v $\mid$ $(\mathit{ed}, u) \leftarrow$ ***rs*** v
table	$:=$	***prev*** $\mid$ ***curr*** $\mid$ ***val***
tc	$:=$	***Fix*** $\mid$ ***Until*** $(\lambda\ g.e)$ $\mid$ ***Iter*** e
f	$:=$	variable representing a function / operator
g	$:=$	variable representing a graph
u, v	$:=$	variable representing a vertex
ed	$:=$	variable representing an edge
fld	$:=$	field name
comb	$:=$	`max` $\mid$ `or` $\mid$ $\cdots$

Figure 6. Core part of Fregel syntax.

Second, Fregel abstracts aggregation and communication by a *comprehension with a specific generator*. Aggregation is described by a comprehension for which the generator is the entire graph (list of all vertices) while communication with adjacent vertices is described by one for which the generator is the list of the adjacent vertices.

These features are the foundation of the functional model discussed in Sect. 2. In addition, Fregel is equipped with four higher-order functions for graphs that provide ways to concisely write various graph computations. Function ***fregel*** corresponds to functional model `pregelModel` shown in Sect. 2. Function ***gzip*** pairs values on every corresponding vertices in two graphs of the same shape, and ***gmap*** applies a given function to every vertex. Function ***giter*** abstracts iterative computation.

Moreover, a Fregel program can be run on Haskell interpreters like GHCi because Fregel's syntax follows that of Haskell. This feature is quite useful for testing and debugging a Fregel program. After testing and debugging, the Fregel program is compiled into a Pregel program for big graph processing. Note that the compiled Pregel program does not handle lazy evaluation.

In the following sections, we first introduce the core part of Fregel's language constructs and then explain Fregel programming by using specific examples. Fregel does not currently allow user-defined recursive functions.

3.2 Fregel Language Constructs

A vertex in Fregel has two lists of edges: a list of incoming edges in the original graph and one of incoming edges in the reversed (transposed) graph. In the rest of paper, 'reversed edge' means an edge in the latter list (an incoming edge in the reversed graph is an edge produced by reversing an outgoing edge in the original graph). The latter list does not exist in the datatype for our functional model described in Sect. 2.1, but we decided to let every vertex have this list to make it easier for programmers to write programs in which part of the computation needs to be carried out on the reversed graph. An example is an algorithm for decomposing a directed graph into its strongly connected components, which is described in Sect. 3.6.

Figure 6 presents the core part of the syntax of Fregel. The tokens in bold-slant font are important reserved words in Fregel.

A Fregel program defines a function that takes a single input graph and returns a resultant graph. An LSS function is abstracted as two functions: the initialization function and the step function. Since an LSS function returns multiple values in many cases, the programmer must often define a record for them ,and each vertex holds the record data. Fregel provides a concise way to access a record field.

The expressions in Fregel are standard ones in Haskell: a combine function applied to a comprehension with specific generators, table access on a vertex v followed by the field selection operator denoted by "`.^`"), and a graph expression with higher-order functions on graphs such as ***fregel***. There are three generators in Fregel: a graph variable to generate all vertices, and ***is*** v / ***rs*** v to generate every pair of v's adjacent vertex u connected by an incoming or reversed edge and the value on this edge.

There are two kinds of definitions in **let**-expressions: ones for a definition of a normal function or constant including an initialization function and ones for a step function, which is the only higher-order function that a user can define. A step function takes two tables, ***prev*** and ***curr***, as well as vertex v, which repeatedly executes the step function. There is another table ***val*** that is used to access the values in the input graph. There are three kinds of termination conditions for higher-order functions ***fregel*** and ***giter***.

3.3 Haskell Implementation of Fregel

As stated in Sect. 3.1, a Fregel program can be run on Haskell interpreters. We thus implemented Fregel as a library of Haskell. Though this Haskell implementation is used only in the testing and debugging phases during the development of a Fregel program, we present here to help the reader understand the behaviors of a Fregel program.

Figure 7 presents the core part of the implementation. The datatypes for graphs are the same as those described in Sect. 2.1 except that each vertex has a list of reversed edges in its record under the field name `rs`. The termination point is defined by the `Termination` type, where ***Fix*** means a steady state, ***Until*** means a termination condition specified by a function, and ***Iter*** specifies the number of iterations of LSS to perform. Function `termination` applies a given termination point to an infinite list of graphs.

The higher-order function ***fregel*** takes as its arguments an initialization function, a step function, a termination condition, and an input graph and returns the resultant graph after its pregel computation. The definition of ***fregel*** here differs somewhat from that of `pregelModel` because it has to implement the memorization mechanism. To do so, ***fregel*** uses two lists of computation results for all vertices, which are accessed via the vertex identifiers.

Function ***gmap*** applies the given function to every vertex in the target graph and returns a new graph with the same shape for which the vertices have the application results. This is simply defined in terms of `makeGraph`.

Function ***gzip*** is given two graphs of the same shape and returns a graph, for which each vertex has a pair of values that correspond to those of the vertices of the two graphs. A pair is defined by the `Pair` type, which has `_fst` and `_snd` fields. This function can be also defined in terms of `makeGraph`.

Function ***giter*** is given four arguments: `init`, `iter`, `term`, and an input graph. It first applies `init` to the input graph and then repeatedly applies `iter` to the result to produce a list of graphs. Finally, it uses `term` to terminate the iteration and obtain the final result. It can be defined by using a standard function, `iterate`.

3.4 Examples: Reachability Problems

Our first example Fregel program is one for solving the all-reachability problem (Fig. 8(a)). Since the LSS function for this problem returns a Boolean value indicating whether the vertex that calls the LSS function is currently reachable or not, we define a record *RVal* that contains only this Boolean value at the *rch* field in this record.

```
data Vertex a b = Vertex { vid :: Int, val :: a,
                           is, rs :: [Edge a b] }

data Termination a =
  Fix | Iter Int | Until (a -> Bool)

data Pair a b = Pair {_fst :: a, _snd :: b}

termination :: Eq a => Termination a -> [a] -> a
termination Fix xs =
  fst . head . dropWhile (\(a,b) -> (a /= b)) $
  zip xs (tail xs)
termination (Iter n) xs = head (drop n xs)
termination (Until p) xs =
  head $ dropWhile (not . p) xs

fregel :: (Vertex a b -> r) ->
          (Vertex a b -> (Vertex a b -> r) ->
             (Vertex a b -> r) -> r) ->
          Termination (Graph r b) ->
          Graph a b -> Graph r b
fregel init step term g =
  let rs0 = map init g
      f rs_old = let rs_new =
                       map (\v -> step v prev curr) g
                     prev u = rs_old !! (vid u)
                     curr u = rs_new !! (vid u)
                 in rs_new
      rss = iterate f rs0
  in termination term (map (makeGraph g) rss)

gmap :: (Vertex a b -> r) -> Graph a b -> Graph r b
gmap f g = makeGraph g (map f g)

gzip :: Graph a1 b -> Graph a2 b ->
        Graph (Pair a1 a2) b
gzip g1 g2 =
  makeGraph
    g1
    (zipWith (\u v -> Pair (val u) (val v)) g1 g2)

giter :: (Eq r, Eq b) => (Vertex a b -> r) ->
         (Graph r b -> Graph r b) ->
         Termination (Graph r b) ->
         Graph a b -> Graph r b
giter init iter term g =
  let g0 = gmap init g
      gs = iterate iter g0
  in termination term gs
```

Figure 7. Haskell implementation of Fregel.

The function *reAll* is the main part of the program, and it defines the initialization and step functions. The initialization function, *init*, returns an *RVal* record in which the *rch* field is *True* only if the vertex is the staring point (vertex identifier is zero). The vertex identifier can be obtained by using a special predefined function, ***vid***. The step function, *step*, collects data at the previous clock from every adjacent vertex connected by an incoming edge. This is done by using the syntax of comprehension, in which the generator is ***is*** *v*. For every adjacent vertex *u*, this program obtains the result at the previous clock by ***prev*** *u* and accesses its *rch* field. Then *step* combines the results of all adjacent vertices by using the *or* function and returns the disjunction of the combined value and its own *rch* value at the previous clock.

```
data RVal = RVal { rch :: Bool } deriving Eq
reAll g =
  let init v = RVal (vid v == 0)
      step v prev curr =
        let newrch =
              prev v .^ rch ||
              or [ prev u .^ rch | (e, u) ← is v ]
        in RVal newrch
  in fregel init step Fix g
```

(a) Program for all-reachability problem

```
data RVal = RVal { rch :: Bool } deriving Eq
re100 g =
  let init v = RVal (vid v == 0)
      step v prev curr =
        let newrch =
              prev v .^ rch ||
              or [ prev u .^ rch | (e, u) <- is v ]
        in RVal newrch
  in fregel init step
       (Until
         (\g -> sum [ 1 | u <- g, val u .^ rch ] > 100))
       g
```

(b) Program for 100-reachability problem

Figure 8. Fregel programs for solving reachability problems.

```
data SVal = SVal { dist :: Int } deriving Eq
sssp g =
 let init v = SVal (if vid v == 0 then 0 else bigNumber)
     step v prev curr =
       let newdist =
             prev v .^ dist `min`
             minimum [ prev u .^ dist + e | (e, u)←is v]
       in SVal newdist
 in fregel init step Fix g
```

Figure 9. Fregel program for single-source shortest path problem.

In `reAll`, `init` and `step` are given to the function ***fregel***. Its third argument, ***Fix***, specifies the termination point, and the fourth argument is the input graph.

Figure 8(b) presents a Fregel program for solving the 100-reachability problem. This program is almost the same as that in Fig. 8(a) except for the termination condition. The termination condition in Fig. 8(b) is ***Until***, which corresponds to `untilValue` in our functional model. ***Until*** takes a function that defines the condition. This function gathers the number of currently reachable vertices by aggregation.

3.5 Example: Single-Source Shortest Path

The next example is similar to the reachability computation, but it uses values on edges to find the shortest path length from the source vertex (0) to every vertex rather than simply detecting whether a vertex is reachable from the source vertex. Figure 9 presents the program. Its structure is the same as that shown in Fig. 8 (a). The LSS function for this problem returns the (tentative) shortest path length to the vertex from the source vertex, so record `SVal` for these return values consists of an integer field, `dist`. The step case of the LSS function uses values on edges, i.e., the first component `e` of the pair generated in the comprehension, to update the tentative shortest path length for a vertex, it takes the minimum of sums of the tentative shortest path length of every neighbor vertex (***prev*** `u .^ dist`) and the edge length (`e`) from it. In general, an edge value can be a tuple of multiple values.

3.6 Example: Strongly Connected Components

As an example of using higher-order functions for graphs, Fig. 10 presents a Fregel program for solving the strongly connected components problem. The output of this program is a directed graph with the same shape as the input graph; the value on each vertex is the identifier of the component to which it belongs. This program is based on the min-label algorithm presented by Yan el al. (Yan et al. 2014).

This program repeats the following four operations until every vertex belongs to a component.

(1) Initialization. Every vertex for which its component have not been found yet sets the `notfound` flag value. This means that the vertex must participate in the following computations.

(2) Forward propagation. Each `notfound` vertex first sets its `minv` value to its own identifier. Then it repeatedly calculates the minimum value of its own (previous) `minv` value and the `minv` values of the adjacent vertices to obtain the minimum value of those that can be propagated to the vertex through the incoming edges. This is repeated until the computation falls into a steady state.

(3) Backward propagation. This is the same as forward propagation except that the direction of `minv` propagation is reversed; each `notfound` vertex updates its `minv` value through the reverse edges.

(4) Component detection. Each `notfound` vertex judges whether the results (identifiers) of forward propagation and backward propagation are the same. If they are, the vertex belongs to the component represented by the identifier.

The program in Fig. 10 has a nested iterative structure.

The outer iteration, which is described in terms of ***giter***, repeatedly performs the above operations for the remaining subgraph until no vertices remain. In this outer loop, each vertex has a record *C* that has only the `sccId` filed. This field has the identifier of the component, which is the minimum identifier of the vertices in the component, or -1 if the component has not been found yet.

In the processing of (1)–(4), each vertex has a record *MN* with two fields. The `minv` field holds the minimum of the propagated values, and the `notfound` field holds the flag value explained above. The initialization uses ***gmap*** to create a graph *ga*. There are two inner iterations: one is to perform forward propagation and the other is to perform backward propagation, both of which are described in terms of ***fregel***. Both propagations take the same graph created in the initialization. Their results, `gf` and `gb`, are combined by ***gzip*** and passed to component detection, which is simply defined by ***gmap***.

Generally, it is difficult to describe a computation that consists of multiple phases of different processing and / or iterative processing as a Pregel program. The four higher-order functions provided by Fregel abstract computations on graphs and thereby enable the programmer to write a program as a combination of these higher-order functions. This functional style of programming in Fregel makes it easier for the programmer to develop a complicated program, like one for solving the strongly connected components problem.

In the following section, we show that a Fregel program described as a combination of the four higher-order functions can be transformed into a Fregel program with a single ***fregel***.

4. Normalization of Fregel Programs

A Fregel program can use the graph higher-order functions multiple times. In a naive compilation of such a program, the underlying Pregel system is invoked once for each use of a graph higher-order function. This is inefficient due to the high cost of starting up the Pregel system. To avoid this problem, we introduce a normalization process that transforms a Fregel program with multiple uses of

```
data MN = MN { minv :: Int, notfound :: Bool } deriving Eq
data C = C { sccId :: Int } deriving Eq
scc g =
  let f_init v = if val v .^ sccId < 0 then MN (vid v) True else MN (val v .^ sccId) False
      f_fw v prev curr =
        let c' = (prev v .^ minv) `min` minimum [ prev u .^ minv | (e,u) <- is v, prev u .^ notfound ]
        in if prev v .^ notfound then MN c' (prev v .^ notfound) else prev v
      f_bw v prev curr =
        let c' = (prev v .^ minv) `min` minimum [ prev u .^ minv | (e,u) <- rs v, prev u .^ notfound ]
        in if prev v .^ notfound then MN c' (prev v .^ notfound) else prev v
      detect v = if val v .^ _fst .^ minv == val v .^ _snd .^ minv then C (val v .^ _fst .^ minv) else C (-1)
      f0 v = val v
      sccInner0 v = C (-1)
      sccInner g = let ga = gmap f_init g
                       gf = fregel f0 f_fw Fix ga
                       gb = fregel f0 f_bw Fix ga
                       gfb = gzip gf gb
                       g' = gmap detect gfb
                   in g'
      gr = giter sccInner0 sccInner Fix g
  in gr
```

Figure 10. Fregel program for solving strongly-connected components problem.

graph higher-order functions into an equivalent Fregel program with a single use of ***fregel***.

A normalized Fregel program uses ***fregel*** with ***Fix*** as its only use of a graph higher-order function in the following form.

```
prog g = let (bindings)...
         in fregel init step Fix g
```

Intuitively, normalization of a Fregel program means building a step function that *emulates* computation of the program. The step function is basically a *phase* transition machine. A phase corresponds to a use of a graph higher-order function in the program, and the step function in the phase runs the original step function used in the graph higher-order function. When the computation of the phase (the graph higher-order function) is finished, the step function moves to the next phase. The emulated computation runs on a *tupled* graph in which each vertex has a tuple of values of all graphs computed in the original program.

As our running example, we will use program `scc` shown in Fig. 10. Its normalized version is shown in Fig. 11.

4.1 Normalization Algorithm

For brevity, we assume (1) that the result of a graph higher-order function is bound to a variable (i.e., A-normal form for graph expressions), (2) that variable names are unique throughout the program, (3) that ***giter***s use different step functions, (4) that user-defined functions are inlined in step functions, and (5) that the types of subexpressions are known. It is easy to satisfy these assumption by using standard techniques.

The normalization process consists of five steps, which are explained in the following sections.

4.1.1 Enumerating Phases

The first step is to enumerate the phases corresponding to uses of graph higher-order functions. By the assumption, this simply means enumerating the variables of graph types. In the following explanation, we will use variables and phases interchangeably.

Let P be the set of phases: $P = \{$ all variables of graph types $\}$. Since ***giter***s need special treatment later, we define a subset of P: $I = \{\, p \mid p \in P,\ p$ binds the ***giter*** result $\}$.

For `scc` in Fig. 10, we have $P = \{\mathit{gr}, \mathit{ga}, \mathit{gf}, \mathit{gb}, \mathit{gfb}, \mathit{g'}\}$ and $I = \{\mathit{gr}\}$.

4.1.2 Building a New Record Type

The next step is to define a new record type, `ND`, for use in the normalized program. Here, $vType(p)$ is the vertex type of a graph bound by variable p.

$$\textbf{data}\ \mathtt{ND} = \mathtt{ND}\ \{\ \mathtt{phase :: Int,\ ss :: Int}\ \} \cup \bigcup_{p\in P}\{\mathtt{dat}_p\ \mathtt{::}\ vType(p)\ \} \cup \bigcup_{p\in I}\{\ \mathtt{ss}_p\ \mathtt{::Int}\ \ \}$$

In the first clause, `phase` is used to hold the current executing phase (represented by an integer) and `ss` is used to count the number of supersteps in the phase. In the second clause, for each phase p, $\mathtt{dat}_p$ is used to hold the result of the computation of phase p. The third clause is for uses of ***giter***, in which $\mathtt{ss}_p$ holds the number of iterations of the ***giter*** bound by p.

The new record data for `scc` is shown at the head of Fig. 11.

4.1.3 Building a Step Function and Termination Judgment for Each Phase

For all phase p in P, the normalized program needs two pieces of code: step function body $\mathtt{step}_p$ for implementing the computation in the phase and termination judgment expression $\mathtt{fexp}_p$ for detecting the end of computation. These are built from components in the expression that p binds in the original program.

During the building process of $\mathtt{step}_p$ and $\mathtt{fexp}_p$, ***curr***, ***prev***, and ***val*** used in the original components must be replaced with suitable counterparts. We define two substitutions, σ_{g_o,g_i} and σ'_g.

$$\begin{aligned}\sigma_{g_o,g_i} &= \{\textbf{\textit{prev}}\,x \mapsto \textbf{\textit{prev}}\,x\,\mathtt{.\hat{}dat}_{g_o}, \textbf{\textit{curr}}\,x \mapsto \textbf{\textit{curr}}\,x\,\mathtt{.\hat{}dat}_{g_o}\} \\ &\quad \cup \sigma'_{g_i} \\ \sigma'_g &= \mathtt{if}\ g \text{ is the input for the whole program} \\ &\quad\ \mathtt{then}\ \{\,\} \\ &\quad\ \mathtt{else}\ \{\textbf{\textit{val}}\,x \mapsto \textbf{\textit{prev}}\,x\,\mathtt{.\hat{}dat}_g\}\end{aligned}$$

Intuitively, these substitutions mean that data access to each graph in the original program is replaced with data access to its corresponding component in the tupled graph in the normalized program.

Since $\mathtt{step}_p$ and $\mathtt{fexp}_p$ depend on the expression p binds, we will explain the building process case by case.

- **Case of** `p =` ***fregel*** `f0 ft tc g`

Here, $\mathtt{step}_p$ performs initialization using the original initialization function `f0` when the number of supersteps in this phase is 0 and then uses `ft` to proceed with the computation afterwards.

```
data ND = ND { phase :: Int, ss :: Int,
               dat_gr :: C, dat_ga :: MN, dat_gf :: MN, dat_gb :: MN, dat_gfb :: Pair MN MN, dat_g' :: C,
               ss_gr :: Int} deriving Eq
scc g =
 let step v prev curr =
      let d_gr = if prev v .^ phase == 1
                 then if prev v .^ ss_gr == 0 then C (-1) else prev v .^ dat_g'
                 else prev v .^ dat_gr
          d_ga = if prev v .^ phase == 2
                 then if prev v .^ dat_gr .^ sccId < 0 then MN (vid v) True else MN (prev v .^ dat_gr .^ sccId) False
                 else prev v .^ dat_ga
          d_gf = if prev v .^ phase == 3
                 then if prev v .^ ss == 0 then prev v .^ dat_ga
                      else
                       let c' = (prev v .^ dat_gf .^ minv) `min`
                                minimum [ prev u .^ dat_gf .^ minv | (e,u) <- is v, prev u .^ dat_gf .^ notfound ]
                       in if prev v .^ dat_gf .^ notfound then MN c' (prev v .^ dat_gf .^ notfound) else prev v .^ dat_gf
                 else prev v .^ dat_gf
          d_gb = if prev v .^ phase == 4
                 then if prev v .^ ss == 0 then prev v .^ dat_ga
                      else
                       let c' = (prev v .^ dat_gb .^ minv) `min`
                                minimum [ prev u .^ dat_gb .^ minv | (e,u) <- rs v, prev u .^ dat_gb .^ notfound ]
                       in if prev v .^ dat_gb .^ notfound then MN c' (prev v .^ dat_gb .^ notfound) else prev v .^ dat_gb
                 else prev v .^ dat_gb
          d_gfb = if prev v .^ phase == 5 then Pair (prev v .^ dat_gf) (prev v .^ dat_gb) else prev v .^ dat_gfb
          d_g' = if prev v .^ phase == 6
                 then if prev v .^ dat_gfb .^ _fst .^ minv == prev v .^ dat_gfb .^ _snd .^ minv
                      then C (prev v .^ dat_gfb .^ _fst .^ minv) else C (-1)
                 else prev v .^ dat_g'
          pend = (prev v .^ phase == 1 && prev v .^ ss_gr > 0 && and [ prev u .^ dat_gr == curr u .^ dat_gr | u <- g ])
                 || (prev v .^ phase == 2 && True )
                 || (prev v .^ phase == 3 && prev v .^ ss > 0 && and [ prev u .^ dat_gf == curr u .^ dat_gf | u <- g ])
                 || (prev v .^ phase == 4 && prev v .^ ss > 0 && and [ prev u .^ dat_gb == curr u .^ dat_gb | u <- g ])
                 || (prev v .^ phase == 5 && True ) || (prev v .^ phase == 6 && True )
          phase' = if pend then next (prev v .^ phase) else stay (prev v .^ phase)
          ss' = if prev v .^ phase  > 0 && prev v .^ phase == curr v .^ phase then prev v .^ ss + 1 else 0
          ss_gr' = if prev v .^ phase == 1 then if pend then 0 else prev v .^ ss_gr + 1 else prev v .^ ss_gr
      in ND phase' ss' d_gr d_ga d_gf d_gb d_gfb d_g' ss_gr'
     init v = ND 1 0 (C 0) (MN 0 False) (MN 0 False) (MN 0 False) (Pair (MN 0 False) (MN 0 False)) (C 0) 0
 in fregel init step Fix g
```

Figure 11. Normalized version of `scc` in Fig. 10. Actual definitions of `stay` and `next` are omitted, but an equivalent definitions are given: `stay i = if i == 1 then 2 else i` and `next i = if i == 1 then 0 else if i == 6 then 1 else i + 1`.

$step_p =$ **if** ***prev*** `v .^ ss == 0` **then** $\sigma_{p,g}$(`f0 v`)
  else $\sigma_{p,g}$(`ft v` ***prev curr***)

Here, substitution expression $\sigma_{p,g}$(`f0 v`) means applying the substitution after inlining the function call `f0 v`. Other substitutions are done in the same manner.

Here, $fexp_p$ depends on the termination condition, `tc`. When `tc` is ***Fix***, judgment is done by checking whether the value of this phase remains unchanged on all vertices during this superstep. Note that this check must be done after running the computation `ft` at least once. Note also that ***curr*** `u .^` dat_p refers to the result of phase p computed in this superstep.

$fexp_p =$
  prev `v .^ ss > 0 &&`
  `and [` ***prev*** `u .^`dat_p `==` ***curr*** `u .^`dat_p `| u <- g ]`

When `tc` is ***Iter*** n, the judgment is done by checking whether the number of supersteps in the current phase is equal to n.

$fexp_p =$ ***prev*** `v .^ ss == n`

Similarly, when `tc` is ***Until***`(\g -> exp)`, the condition expression becomes the judgment.

$fexp_p = \sigma'_g(exp)$

- **Case of** `p = ` ***gmap*** `f g`
  Here $step_p$ simply applies the original function `f`. Since ***gmap*** does not need to be iterated, the termination judgment, $fexp_p$, is always true.

  $step_p =$ $\sigma_{p,g}$(`f v`)
  $fexp_p =$ `True`

- **Case of** `p = ` ***gzip*** `g1 g2`
  Similar to the ***gmap*** case, here $step_p$ simply pairs components corresponding to two graph variables `g1` and `g2`.

  $step_p =$ `Pair (`***prev*** `v .^`dat_{g1}`) (`***prev*** `v .^`dat_{g2}`)`
  $fexp_p =$ `True`

- **Case of** `p = ` ***giter*** `f0 ft tc g`
  Here, $step_p$ performs initialization by using `f0` for the first arrival at this phase and copies values computed by `ft` afterwards. Note that $step_p$ uses a specific counter ss_p (the number of arrivals at this phase) instead of the counter `ss` used in other cases. Here, *outputOf*(`ft`) is the output variable of `ft`.

  $step_p =$ **if** ***prev*** `v .^`ss_p `== 0` **then** $\sigma_{p,g}$(`f0 v`)
    else ***prev*** `v .^`$dat_{outputOf(ft)}$

Similar to ***fregel***, the termination judgment expression, $fexp_p$, depends on the termination condition, `tc`. The difference is that

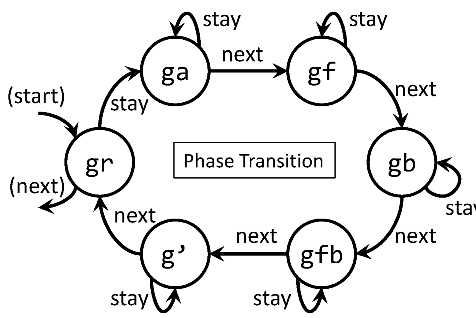

Figure 12. Phase transition machine for *scc*.

the specific counter ss_p is used instead of *ss*. When *tc* is ***Fix***, the judgment is as follows.

```
fexp_p =
  prev v .^ss_p > 0 &&
  and [ prev u .^dat_p == curr u .^dat_p | u <- g ]
```

Similarly, other cases are as follows.

tc = ***Iter*** n ⇒ $fexp_p$ = ***prev*** `v.^ss_p== n`
tc = ***Until*** `(\g -> exp)` ⇒ $fexp_p = \sigma'_g(\mathit{exp})$

4.1.4 Building a Phase Transition Machine

Now we define a phase transition machine by using two functions: One is *next* :: $P \to P$ to indicate what phase will be executed next when the computation of the current phase terminates (i.e., when the termination judgment expression returns *True*), and the other is *stay* :: $P \to P$ to indicate what phase will be executed to continue the computation in the current phase (i.e., the *False* case). Basically, *stay* $p = p$ for most phases, but for phases binding ***giter***s, *stay* returns another phase because the computation in such a ***giter*** phase consists of a chain of other phases.

Here, *next* is defined as the union of topological sorts done using the dependencies of graph variables in each graph function (i.e., function receiving a graph to return a graph; being the whole program or used by ***giter***s). In addition, $next(outputOf(\texttt{ft})) = p$ when p binds a ***giter*** expression, i.e., *p* = ***giter*** *f0 ft tc g*.

Here, *stay* is basically defined by *stay* $p = p$. An exception is that, for *p* = ***giter*** *f0 ft tc g*, it is defined as $stay(p) =$ the first variable in the topological sort in *ft* .

For example, *next* and *stay* for *scc* are given as follows. From the topological sort in *sccInner*, we have $next(ga) = gf$, $next(gf) = gb$, $next(gb) = gfb$, $next(gfb) = g'$. Since *gb* and *gf* have no dependency, we can swap them. In addition, because *gr* binds a ***giter*** expression, we have $next(g') = gr$. Similarly, we have $stay(gr) = ga$. The rest of *stay* is given by $stay(p) = p$. Figure 12 shows phase transition using these functions.

4.1.5 Building a Normalized Program

A normalized program is built by using the components built so far. For simplicity, let phases $p \in P$ be represented by different integers in $\{1, \ldots, |P|\}$ and introduce a special phase 0 to indicate the end of computation. In addition, let $stay(0) = 0$ and $next(g) = 0$ for the output graph variable g in the original program, and also let the ***giter*** phase set I be $\{i_1, \ldots, i_{|I|}\}$.

Figure 13 shows a template of a normalized program. The main part is the step function to emulate the original computation. It runs one of the step function bodies $step_p$ in accordance with the current phase ***prev*** *v* .^ *phase*. The phase transition is controlled by the termination judgments, $fexp_p$, and the transition functions, *next* and *stay*. Note that the step function returns the same value as the input once ***prev*** .^ *phase* becomes 0, and then the computation terminates. The initialization function *init* simply initializes the current phase *phase* to 1, the counters *ss*, $ss_{i_1}', \ldots, ss_{i_{|I|}}'$ to 0, and the other components for graph variables to their default values (e.g., 0 for integer components). The normalized program in Fig. 11 is an instance of the template.

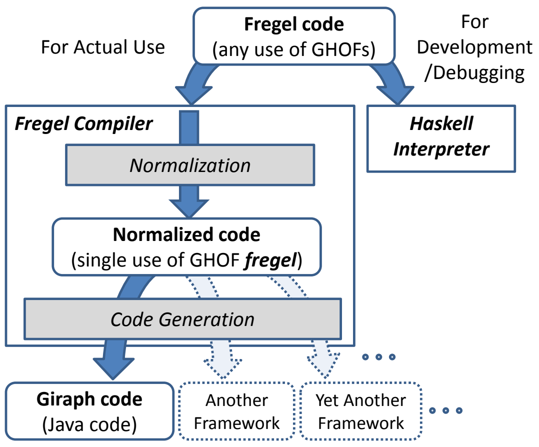

Figure 14. Flows of Fregel program compilation and interpretation (GHOF: graph higher-order function).

4.2 Optimization in Normalization

For brevity, the transformation explained so far disregards the efficiency of the normalized program and introduces much redundancy. We can perform optimization to reduce the redundancy.

An easy optimization deals with redundancy related to the emulation of ***gzip***s. Instead of creating a pair to emulate ***gzip***, we can supply its components directly to its consumers. For example, in the normalized program in Fig. 11, ***prev*** `.^ dat_gfb .^ _fst` and ***prev*** `.^ dat_gfb .^_snd` in line 28 can be replaced with ***prev*** `.^ dat_gf` and ***prev*** `.^ dat_gb` , respectively.

Another simple optimization is to fuse a ***gmap*** into another higher-order function when it is the only consumer of the ***gmap***'s output. These are typical fusion transformations in functional programming. They can reduce the size of the tupled graph and the number of phases (supersteps), thereby improving the efficiency of the normalized program.

In addition, we can run multiple independent phases simultaneously to reduce the total number of supersteps although the phase-transition machine explained so far runs only one phase at a time for simplicity. In the BSP model, the number of supersteps has the greatest effect on efficiency. For example, *gf* and *gb* in *scc* are independent, so they can be run simultaneously.

5. Compiling Fregel Programs into Giraph Code

We have developed a prototype system for compiling Fregel programs into Giraph code (Giraph is a widely-used open-source Pregel-like framework). In this section, we briefly describe how the normalized Fregel program is compiled into a Giraph program. Since we do not use any Giraph-specific features in our prototype system, we can implement it in other Pregel-like frameworks. The whole flow of the Fregel compiler is shown in Fig. 14.

After normalization in the previous section, the normalized program is compiled using the following steps.

1. Unwind the (circular) dependencies
2. Split an LSS function into physical supersteps
3. List the fields for vertices

```
prog g = let step v prev curr =
               let d_1 = if prev v .^ phase == 1 then step_1 else prev v .^ dat_1
                   ··· — snip
                   d_|P| = if prev v .^ phase == |P| then step_|P| else prev v .^ dat_|P|
                   pend = (prev v .^ phase == 1 && fexp_1) || ··· || (prev v .^ phase == |P| && fexp_|P|)
                   phase' = if pend then next (prev v .^ phase) else stay (prev v .^ phase)
                   ss' = if prev v .^ phase  > 0 && prev v .^ phase == curr v .^ phase then prev v .^ ss + 1 else 0
                   ss_i1' = if prev v .^ phase == i_1 then if pend then 0 else prev v .^ ss_i1 + 1 else prev v .^ ss_i1
                   ··· — snip
               in ND phase' ss' d_1 ··· d_|P| ss_i1' ··· ss_i|I|'
             init v = ND 1 0 ... 0 ··· 0
         in fregel init step Fix g
```

Figure 13. Template of normalized Fregel program.

4. List the comprehensions and convert them to communication
5. Generate code for the `vertex.compute` function

5.1 Unwind Dependencies

As we mentioned in Sect. 3.1, the syntax of Fregel follows that of Haskell. Since the compiled Giraph program (written in Java) is not evaluated in a lazy manner, we first rearrange the let-bindings so that we can evaluate them in order.

As a running example of the compilation, let us consider the following fragment of a Fregel program. The ***curr*** table is used here to access the vertex data at the current clock. Note that the ***curr*** table could be introduced in the normalization of Fregel programs even though user-defined Fregel programs do not have it.

```
data RVal = RVal { rch :: Bool, spl :: Bool }
 ...
let newv = prev v .^ rch || neighbor
    neighbor =
      or [prev u .^ rch | (e, u) <- is v]
    aggv = and [curr u .^ rch | (e, u) <- g]
    supplement = neighbor && aggv
in  RVal { newv, supplement }
```

Thanks to the lazy evaluation mechanism in Haskell, this program runs correctly although the value ***curr*** `u .^ rch` is set in the last constructor. To make this program executable Without lazy-evaluation mechanism, we decompose the constructor into substitutions and arrange the let-bindings. The result of this rearrangement is as follows.

```
neighbor = or [prev u .^ rch | (e, u) <- is v]
newv = prev v .^ rch || neighbor
curr v .^ rch = newv
aggv = and [curr u .^ rch | (e, u) <- g]
supplement = neighbor && aggv
curr v .^ spl = supplement
```

5.2 Split LSS Function into Physical Supersteps

The comprehensions in Fregel programs correspond to communication via messages in Giraph programs. Under the restriction of the BSP model that the messages are available only in the next superstep, we cannot compute some value and use it through communication in a single superstep. Such a situation happens when Fregel programs access to the ***curr*** table, so we split the LSS function into multiple supersteps. We call them *physical* supersteps (PSSs) to clarify the superstep in Giraph programs.

In our running example, the value ***curr*** `v .^ rch` is set and then used in the following comprehension. Therefore, we split the LSS function into two physical supersteps as follows.

```
— PSS1
neighbor = or [prev u .^ rch | (e, u) <- is v]
newv = prev v .^ rch || neighbor
curr v .^ rch = newv

— PSS2
aggv = and [curr u .^ rch | (e, u) <- g]
supplement = neighbor && aggv
curr v .^ spl = supplement
```

Note that the variable `neighbor` is defined in PSS1 and used in PSS2. In the following step, we will add those variables that are used over physical supersteps as the fields of vertices to remember the values.

5.3 List the Fields for Vertices

In the third step, we list the fields that a vertex should have. A vertex should have the following fields to compute the compiled Giraph program:

- phase (`v.phase`),
- step count in phase (`v.ss`),
- iteration count for each phase that comes from ***giter***,
- fields of records in the Fregel program, and
- variables that are used over physical supersteps.

5.4 List the Comprehensions and Convert them to Communication

In the fourth step, we list up the comprehensions and convert them as follows.

- For each comprehension with a generator for incoming edges (***is***) or reversed edges (***rs***), we add message senders in the previous superstep and message handlers in the current superstep.
- For each comprehension with a generator for the entire graph, we add an aggregation call in the previous superstep and its reader in the current superstep.

In our running example, we have two comprehensions: one is converted into message sending and handling, and the other is converted into aggregation.

```
— PSS0
sending prev u .^ rch through outgoing edges

— PSS1
neighbor = or [messages]
newv = prev v .^ rch || neighbor
curr v .^ rch = newv
aggregate curr u .^ rch

— PSS2
aggv = getAggregatedValue
supplement = neighbor && aggv
curr v .^ spl = supplement
```

Table 1. Results on a PC cluster

	execution time (s)						number of	total
	P=1	P=2	P=4	P=8	P=12	P=16	supersteps	messages (MB)
Single-source shortest path (rand-s)	294.8	137.4	63.53	35.33	25.40	26.86	58	4,762
Single-source shortest path (rand-d)	N/A	N/A	391.8	192.1	150.8	135.1	50	40,810
Densest subgraph (rand-s)	95.94	52.11	31.17	19.60	16.16	13.91	185	389
Strongly connected components (rand-s)	404.7	160.0	73.07	41.00	30.53	28.32	58	4,622

In Giraph, and most Pregel-like systems, we should specify a single class for the messages. After listing all the messages, we generate a class that consists of a tag and those message values.

5.5 Generate Code for the `vertex.compute` Function

Finally, we generate the `vertex.compute` function of Giraph programs. The outline of the generated function is as follows.

```
public void compute(Vertex<...> vertex,
                    Iterable<MyMessage> msgs) {
  MyVertex v = vertex.getValue();
  if (getSuperstep() == 0) {
    // initializing phase and step
    v.phase = new IntWritable(0);
    v.ss = new IntWritable(0);
  }
  switch (v.phase.get()) {
  case 0: {
    switch (v.ss.get()) {
    case 0: {
      // code for init-function for phase 0
      v.ss = new IntWritable(1);
    } break;
    case 1: {
      // code for judging termination
      if (...) { vertex.voteToHalt(); return; }
      // code for phase transition
      if (...) { v.phase = new IntWritable(1);
                 return; }
      ...
    } break;
    case 2: {
      // handling (received) messages
      for (MyMessage msg : msgs) { ... }
      // computation in LSS function
      ...
      // sending message for next superstep
      // update v.ss
    } break;
   ...
  }
  vertex.setValue(v);
}
```

After the preparation described in Sections 5.1–5.4, each line in the Fregel program is basically translated into a few lines of Java programs.

6. Evaluation

6.1 Performance and Overhead on a PC Cluster

To evaluate the performance of the code compiled from Fregel programs, we conducted several experiments on a PC cluster for three target problems.

- Single-source shortest-path problem (Figure. 9)
- Densest subgraph problem: for this problem (Bahmani et al. 2012), it is easy to write a sequential program and nontrivial to write a Pregel program.
- Strongly connected components problem (in Sect. 3.6)

The data we used were two randomly generated graphs: "rand-s" had $|V| = 1 \times 10^6$ vertices and $|E| = 1 \times 10^7$ edges, and "rand-d" had $|V| = 1 \times 10^6$ vertices and $|E| = 1 \times 10^8$ edges.

The hardware environment of a PC cluster consisted of 16 PCs connected by Gigabit Ethernet, where each PC had an Intel Core i5 CPU (Core i5-2500 in 9 PCs and i5-760 in 7 PCs), 8 GB of memory, and a 128 GB SSD. The versions of the OS, Java, Hadoop, and Giraph were Ubuntu 14.04.3 LTS, 1.8.0_66, 0.20.203.0, and 1.2.0-SNAPSHOT, respectively.

Table 1 shows the execution times excluding the initial arrangement of vertices and output, the number of supersteps, and the total size of messages for each run. These numbers are the mediums among five runs for each set of parameters. P is the number of worker nodes involved in the parallel computation. All these compiled code ran in a reasonable time, and resulted in parallel speedups with the increase of worker nodes.

We expected factors of the overheads of compiled code to be (1) the larger vertices, (2) the more message communication, (3) the increase of supersteps, and (4) missing `voteToHalt` during the computation. To analyze the overhead of generated code from Fregel, we also executed several hand-written programs for the single-source shortest-path problem and the densest subgraph problem.

For the single-source shortest-path problem, we used the following six programs: (a) a hand-written code similar to that in the original Pregel paper (Malewicz et al. 2010), (b) the same algorithm as the hand-written code except for the generated class for vertices, (c) the same algorithm as the hand-written code except for the generated classes for vertices and messages, (d) a modified algorithm so that it has the same number of supersteps as the compiled code, (e) the modified algorithm that uses no `voteToHalt` until reaching a fixpoint, and (f) the modified algorithm in which all the vertices always send messages to neighbors. Table 2 shows the execution times, number of supersteps, and the total size of messages for these programs. We can see from the difference between (b) and (c), and between (e) and (f), that the total size of messages is the main overhead of the generated code from Fregel. The increase of supersteps between (c) and (d) also slows down the computation (20–40% in this case). Compared to these overheads, the larger vertices and the missing `voteToHalt` affect the performance only a little.

For the densest subgraph problem, we used the following four programs. (a) a carefully hand-written code so that the number of supersteps and messages are minimized[2]; (b) the same algorithm as the hand-written code except for the generated class for vertices; (c) the same algorithm as the hand-written code except for the generated class for vertices and messages; (d) a modified algorithm so that it has almost the same number of supersteps as the compiled code. Table 2 shows the execution times, number of supersteps, and the total size of messages for these programs. We can also see in these results that the size of messages affects the performance a lot. In addition, the difference between (c) and (d) for the increase of supersteps and the difference between (d) and Fregel are not negligible: we consider the reason of these overheads is in quite a

[2] This does not utilize `voteToHalt` during the computation. We require a mechanism that activates all the inactive nodes to utilize `voteToHalt`.

Table 2. Overhead analysis for single-source shortest path

	execution time (s)		super-	mes-
	P=4	P=16	steps	sage
a. hand-written	7.69	4.23	29	413
b. + gen. vertex	7.84	4.39	29	413
c. + gen. message	16.35	6.67	29	413
d. + twice supersteps	19.10	9.32	58	413
e. + no VoteToHalt	18.62	9.21	58	413
f. + always sending	59.98	26.00	58	4,762
Fregel	63.53	26.86	58	4,762

Table 3. Overhead analysis for densest subgraph

	execution time (s)		super-	mes-
	P=4	P=16	steps	sage
a. hand-written	14.12	8.14	125	222
b. + gen. vertex	15.29	8.99	125	222
c. + gen. message	18.23	9.54	125	389
d. + 1.5 $\times$ supersteps	23.29	13.21	187	389
Fregel	31.17	13.91	185	389

Table 4. Execution time (s), number of supersteps and total message size for single-source shortest path on Amazon EMR

	P=8	P=16	P=24	P=36	P=48
Hand-written	341.3	152.1	106.3	77.8	75.7
Fregel	3208	2887	999.6	656.1	598.5

Hand-written:	no. of supersteps = 42
	total message size = 25.6 GB
Fregel:	no. of supersteps = 84
	total message size = 348.8 GB

few conversions between primitive values (type `double`, etc.) and Hadoop-specific values (type `DoubleWritable`, etc.).

6.2 Performance on Amazon EMR Cloud Environment

We also made bigger experiments over Amazon EMR cloud environments. The environment of computation nodes was m3.xlarge (CPU Intel Xeon E5-2670 v2, Memory 15 GB, SSD), running Java 1.7, Hadoop 1.0.3, and Giraph 1.2.0-SNAPSHOT.

Table 4 shows the execution times for a randomly generated graph with $|V| = 5 \times 10^7$ vertices and $|E| = 5 \times 10^7$ edges. The compiled code has overhead (by a factor of 8–10) but it shows reasonable parallel speedups even on this large cloud environment.

7. Related Work

Many researches have investigated recursive approaches to programming graph algorithms in functional languages. There are three approaches. The first approach is to represent cyclic and shared structures using structured trees with binders and then to define structural recursion over this new representation as a basic component for processing graphs (Fegaras and Sheard 1996; Hamana 2010; Oliveira and Cook 2012). The second approach is to use inductive but unstructured representations of graphs. Erwig (Erwig 1997, 2001) proposed an inductive representation with two constructors: an empty graph constructor (the base case) and a graph extended with a node together with its label and edges (the inductive case). *Active pattern matching* is used to define graph algorithms in a recursive way. The third approach is to use infinite regular trees to simulate graphs so that graph algorithms can be specified as structure recursions on infinite regular trees (Buneman et al. 2000; Hidaka et al. 2010, 2013). However, all these approaches focus on sequential computation of graphs while we provide a functional approach to parallel graph processing.

There has been much work on DSL for big graph processing on Pregel. Green-Marl (Hong et al. 2012) is an imperative DSL with built-in data types, operators and functions tailored for the implementation of graph algorithms. Hong et al. showed (Hong et al. 2014) that Green-Marl can be compiled into Pregel. Our approach is different in that our DSL is based on a new functional model and structured by structural recursive functions. Palovca (Lesniak 2012) is an embedded DSL using Haskell as the underlying host language for specifying Pregel algorithms. Unlike our DSL, which is based on a new functional model, it uses a monad to hide the side effects and defines the vertex program in a lower-level way like that in Giraph, which is less declarative than ours. Tung and Hu (Tung and Hu 2015) proposed a DSL for querying graphs on the basis of structural recursion and showed how to compile them into Pregel. However, the language is limited to querying. Moreover, as it is based on tree simulation of graphs, it cannot deal with aggregations on graphs.

It is also useful to design skeletons for describing graph algorithms as they can not only capture a certain class of computation pattern but also be implemented efficiently. One representative work is by Launchbury (Launchbury 1995), which showed that a depth-first-search (DFS) computation pattern can be abstracted as a higher-order function in Haskell and used to specify and reason about many interesting graph algorithms.

The GAS (gather-apply-scatter) model (Bae and Howe 2015; Gonzalez et al. 2012; Sengupta et al. 2015) has recently emerged as a realistic parallel computational model for big graphs. The GAS model and our Fregel model have the same structure of communication (i.e., communication only between neighbor vertices), but their styles of computation differ. The GAS model allows asynchronous computation among vertices to obtain realistic performance by restricting the form of computation in a vertex while the Fregel model allows only synchronous computation. It would be interesting to investigate when and how we can compile Fregel code for asynchronous computation to obtain realistic performance.

8. Conclusion

We have presented a novel functional DSL for supporting the development of Pregel programs for processing big graphs. It resembles the role of Pig (Gates et al. 2009) and Hive (Thusoo et al. 2009) systems for supporting development of MapReduce programs from a high-level specification for processing big key-value data. Different from the imperative Pregel, Fregel is functional and declarative, completely removing message passing and explicit state control. The results for compiled Pregel programs generated by our prototype system (which has much room for further optimization) are quite encouraging; the programs have good speedup, are scalable, and are only five to ten times slower than carefully handwritten programs although the system does no optimization.

Fregel's current absolute performance is not sufficient for practical systems for big graph processing. Further work is needed to analyze the performance weaknesses and to develop optimization methods to resolve them. In addition, we will study formal proofs of the correctness of the compilation and normalization and develop sophisticated optimization methods on top of the formal proofs.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 15K15974 and 26280020.

References

S.-H. Bae and B. Howe. GossipMap: A distributed community detection algorithm for billion-edge directed graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 27:1–27:12. ACM, 2015.

B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and MapReduce. *Proceedings of the VLDB Endowment*, 5(5):454–465, 2012.

P. Buneman, M. Fernandez, and D. Suciu. UnQL: A query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

M. Erwig. Functional programming with graphs. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 52–65. ACM, 1997.

M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 284–294. ACM, 1996.

A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30. USENIX Association, 2012.

M. Hamana. Initial algebra semantics for cyclic sharing tree structures. *Logical Methods in Computer Science*, 6(3):1–23, 2010.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 205–216. ACM, 2010.

S. Hidaka, K. Asada, Z. Hu, H. Kato, and K. Nakano. Structural recursion for querying ordered graphs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 305–318. ACM, 2013.

S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362. ACM, 2012.

S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 208–218. ACM, 2014.

J. Launchbury. Graph algorithms with a functional flavour. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 308–331. Springer-Verlag, 1995.

M. Lesniak. Palovca: Describing and executing graph algorithms in haskell. In *Proceedings of the 14th International Conference on Practical Aspects of Declarative Languages*, PADL'12, pages 153–167. Springer-Verlag, 2012.

Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8): 716–727, 2012.

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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2):25:1–25:39, 2015.

B. C. Oliveira and W. R. Cook. Functional programming with structured graphs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 77–88. ACM, 2012.

D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. GraphReduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12. ACM, 2015.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a Map-Reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

L. D. Tung and Z. Hu. Towards systematic parallelization of graph transformations over Pregel. In *Proceedings of the 8th International Symposium on High-level Parallel Programming and Applications*, HLPP '15, 2015.

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.