複数の頂点主体グラフ計算フレームワーク向けの中間表現とコード生成器

松崎 公紀　岩崎 英哉　江本 健斗　胡 振江　森畑 明昌

著者らは，大規模グラフ並列処理のための関数型領域特化言語 Fregel を提案し，そのコンパイラを作成している．その最初の実装では，頂点主体グラフ計算フレームワークのひとつ Giraph 上で動作するプログラムを生成していた．生成されたコードが，Giraph の他 Pregel+など複数の頂点主体グラフ計算フレームワーク上で利用できることは，領域特化言語の重要な利点である．現在，複数のフレームワークを対象としたコード生成器に向けて，コンパイラの改良を行っている．本論文では，頂点主体グラフ計算に対するフレームワーク中立な中間表現の設計，中間表現へのプログラム変換と中間表現からのコード生成の実装について報告する．

1　はじめに

近年，グラフやネットワーク構造を持つ大規模データ (以後「大規模グラフデータ」と呼ぶ) を処理対象とし，これらのデータから新しい知識を獲得する需要が増している．典型的な大規模グラフデータの例としては，ソーシャルネットワーク，文献参照関係等の書誌情報ネットワーク，生命システムの情報ネットワークなどがあげられる．このようなアプリケーションの開発支援に共通する一般的な課題は，次のような点である．

Kiminori Matsuzaki, 高知工科大学, Kochi University of Technology.
Hideya Iwasaki, 電気通信大学, The University of Electro-Communications.
Kento Emoto, 九州工業大学, Kyushu Institute of Technology.
Zhenjiang Hu, 国立情報学研究所, National Institute of Informatics.
Akimasa Morihata, 東京大学, The University of Tokyo.

* Intermediate Representation and Code Generator for Frameworks for Vertex-centric Graph Computation

This is an unrefereed paper. Copyrights belong to the Authors.

- 多くのアプリケーションに共通して現れる計算構造の簡潔な記述を可能とすること．
- 近年のグラフデータの巨大化に対応し，効率の良い並列処理を可能とすること．

グラフデータは，実世界の多くの場面で自然に現れるため，これらの課題を解決することは極めて重要である．

我々は，大規模グラフデータに対するスケーラブルな分散並列処理のためのモデルとして，Google による Pregel [5] に注目している．Pregel の大きな特徴は，バルク同期並列 [7] に基づく頂点主体の計算モデルを採用している点である．このモデルでは，グラフ内の頂点はクラスタ中の計算ノードに分散され，各頂点は必要に応じて互いにメッセージを交換しながら，スーパーステップ (superstep) と呼ばれる一単位の処理を繰り返し実行する．ひとつのスーパーステップにおいては，各頂点は (共通の) ユーザ定義関数を (並列に) 呼び出す．このユーザ定義関数は，スーパーステップにおける頂点の振舞いを記述しており，典型的には，ひとつ前のスーパーステップにおいて自分に送られたメッセージを受信する，計算を行いその結果に基づいて当該頂点の状態を変更する，別頂点にメッセージを送信する (このメッセージは次のスーパーステップにおいて送信先頂点が受け取る)，各頂点の持

つ情報をグラフ全体で集約 (aggregate) して大域的な情報を得る，などの動作を行う．計算の実行過程において，各頂点は実行状態か停止状態のいずれかをとる．スーパーステップにおいて共通のユーザ定義関数を呼ぶのは，実行状態の頂点だけ (したがって集約に参加するのも実行状態の頂点だけ) である．実行状態の頂点が停止状態に移行するためには，`voteToHalt()` 関数を明示的に呼ぶ必要がある．停止状態の頂点に対してメッセージが送られると，次のスーパーステップにおいてその頂点は実行状態となり，メッセージを受け取り計算を行う．あるスーパーステップにおいて，全頂点が停止状態になりかつ送信されたメッセージがなければ，全体の計算が終了する．

頂点主体のバルク同期並列に基づく Pregel は，プログラマはデータフローや同期のことを考える必要がないという意味で自然なプログラミングモデルを提供する．しかし，実際にスーパーステップに基づくプログラミングを行おうとすると，以下のような煩雑さに直面することとなる [3] [9] [2].

- 明示的な副作用に依存したプログラミング：グラフ上の計算は，各頂点の持つ値の局所的な更新や，他頂点への明示的なデータ送信によって行われる．そのため，ユーザ定義関数を越えた副作用に依存したプログラミングを行う必要がある．
- 明示的な状態制御：頂点間のメッセージ送受信では，ひとつ前のスーパーステップで送信された値が次のスーパーステップで受信される．大域的な情報を得るための集約でも，集約された値は次のスーパーステップでしか利用できない．そのため，メッセージ送受信や集約の前後で計算処理を分断し,（ひとつの）ユーザ定義関数の中で計算処理の遷移を明示的に扱う必要がある．分断された計算処理が増えて複雑になるほど，各時点において送受信されるメッセージの意味やプログラムの流れを把握することが難しくなる．
- 明示的な計算停止制御：プログラムを停止するには，すべての頂点を停止状態にする必要がある．その際，頂点を停止状態へと移行させる `voteToHalt()` は，計算の順番を正しく把握して適切な場所に挿入する必要がある．

これらの問題を解決するため，我々は頂点主体のグラフ処理を記述するための関数型領域特化言語 Fregel [3] [9] を提案，実装している．Fregel は，関数型言語 Haskell のサブセットとなっており，頂点主体の計算を行うプログラムの宣言的な記述を可能としている．Fregel を用いることにより，プログラマは上で述べたようなプログラミングの煩雑さから解放されることが期待されている．

Fregel のコンパイラは，Fregel プログラムから，既存の頂点主体分散並列グラフ処理フレームワークのプログラムへの翻訳系である．Fregel を提案した時点では，Pregel のオープンソース実装のひとつである Giraph [1] 上で動作するプログラムを生成できるだけであった．現在，Giraph 以外にも Pregel+ [8] など複数のフレームワークを対象としたコード生成器に向けて，コンパイラの改良を行っている．この改良では，複数のフレームワークへの翻訳をモジュラに行うことを可能とするため，ユーザが記述した Fregel プログラムを正規化したあと，フレームワーク独立な中間表現にいったん変換する．この共通の中間表現を経由することで，各フレームワーク向けのプログラム生成をより容易に行うことを狙う．

本稿では，頂点主体グラフ計算に対するフレームワーク独立な中間表現 FregelIR (Fregel Intermediate Representations) の設計，および，Fregel プログラムから中間表現への変換と中間表現からのコード生成の実装について報告する．

2 Fregel

2.1 概要

Fregel [3] [9] では，頂点主体の計算を「グラフ高階関数」と呼ばれる高階関数群を用いて記述する．代表的なグラフ高階関数は関数 `fregel` であり [†1]，次の形をしている．

`fregel` *init step term graph*

ここで *init* は各頂点における初期化関数，*step* は各頂点における (論理的な) 反復処理 1 回分を記述する関数，*term* は計算の終了条件，*graph* は処理対象の

†1 グラフ高階関数には他に `gmap`，`gzip`，`giter` がある．

```
 1  data SVal = SVal { dist :: Int }
 2              deriving (Eq, Show)
 3  sssp g =
 4    let init v =
 5          SVal (if vid v == 1 then 0 else 999)
 6        step v prev curr =
 7          let newdist =
 8                prev v .^ dist `min`
 9                minimum [ prev u .^ dist + e |
10                           (e, u) <- is v ]
11          in SVal newdist
12    in fregel init step Fix g
```

図 1　単一始点最短路問題のための **Fregel** プログラム

グラフである．*step* は，頂点自身と他の頂点の 1 回前の計算結果をもとに計算を行い，新しい頂点の値を返すものである．このような *step* の内容を「論理スーパーステップ (logical superstep，以下 LSS)」と呼ぶ [†2]．LSS の処理の中には，隣接頂点の値を参照することや，集約処理を用いて大域的な情報を得ることがあるかもしれない．そのようなときには，LSS 1 回分は，Pregel における複数のスーパーステップから構成されることになる．

LSS 関数 *step* には，隣接頂点間のメッセージ送受信を明示的には記述しない．LSS 関数の引数は，自頂点 `v`，他頂点における前回の LSS における値を保持する表 `prev` である [†3]．また，`is v` によって，自分への入力辺の重み `e` とその辺で結ばれた隣接頂点 `u` の対 `(e,u)` のリストを得ることができる．1 回前の LSS において隣接頂点 `u` が持つ値は `prev u` として得ることができる．

Fregel の簡単なプログラム例として，単一始点最短路問題 (single-source shortest path，以下 SSSP) を解くプログラムを，図 1 に示す．ここでは，頂点番号が 1 である頂点を始点とする．頂点 `v` の頂点番号は，プリミティブ関数 `vid` を用いて得る．

各頂点は，現時点で得られている自身までの最短路を，レコード `SVal` (1 行目) のフィールド値 `dist` に保持する．レコード中のフィールド値は，二項演算子

†2　*step* は頂点における LSS の内容を関数として定義したものであるため，*step* のことを LSS 関数とも呼ぶ．

†3　実際にはもう一つ `curr` と呼ばれる表も引数にとるが，後で述べる正規化の結果で使われてこの段階では用いられないので，ここでは説明しない，

```
 1  data SVal = SVal { dist :: Int }
 2              deriving (Eq, Show)
 3  data MVal = MVal { mval :: Int }
 4              deriving (Eq, Show)
 5  ssspmaxv g =
 6    let ssspinit v =
 7          SVal (if vid v == 1 then 0 else 999)
 8        ssspstep v prev curr =
 9          let newdist =
10                prev v .^ dist `min`
11                minimum [ prev u .^ dist + e |
12                           (e, u) <- is v ]
13          in SVal newdist
14        maxvinit v = MVal (val v .^ dist)
15        maxvstep v prev curr =
16          let newmval =
17                prev v .^ mval `max`
18                maximum [ prev u .^ mval |
19                           (e, u) <- is v ]
20          in MVal newmval
21        g1 = fregel ssspinit ssspstep Fix g
22        g2 = fregel maxvinit maxvstep Fix g1
23    in g2
```

図 2　グラフの直径を求める **Fregel** プログラム

`.^` を用いて取得することができる．

4 行目からの `init` は初期化関数であり，その引数は頂点である．頂点番号が 1 の時には，得られている最短路を 0 とし，そうでなければ無限大 (ここでは 999 としている) とする．

6 行目からの `step` が LSS 関数であり，計算の主要部分である．`step` では，`is v` を生成子とする内包表記を用いて，入力隣接頂点の直前の LSS における計算値のレコードを `prev u` によって取得し，その中のフィールド `dist` に辺の重み `e` を加えて自身までの距離とし，それらの最小値を `minimum` を用いて求める．さらに，この値と，自身の直前の LSS における計算値，すなわち現状得られている最短路 `prev v` との最小値を `newdist` として，`SVal newdist` を `step` の値とする．この値は次の LSS における `step` の中で，テーブル `prev` によって参照可能となる．

12 行目で，上で定義した初期化関数，LSS 関数などを引数にして `fregel` の呼び出している．第 3 引数の `Fix` は，LSS の反復により頂点の保持する値 (`SVal` 型) がすべて変化しなくなったらプログラムを停止することを指定している．最終結果は，各頂点が `SVal` 型の値 (`dist` に最短路長がある) を持つような，新しいグラフである．

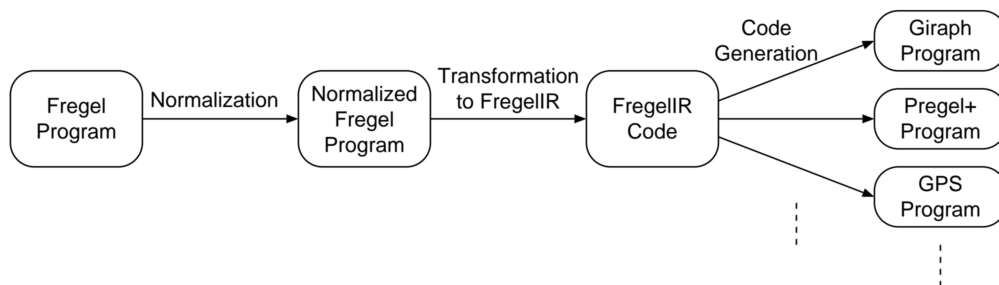

図 3　Fregel プログラムのコンパイル処理の流れ

グラフ高階関数 `fregel` を複数回利用する簡単なプログラム例として，ある始点からの最短路の最大値，すなわちその始点を端点とするグラフの直径を求めるプログラムを，図 2 に示す．直径を求めるために，全頂点は SSSP のようにして始点からの最短路を求めた後，再度 `fregel` を用いてそれらの最大値を求める (この部分を MAXV と呼ぶ) こととする．MAXV の `fregel` による計算は，SSSP とほとんど同じ構造をしているので，説明を省略する．このプログラムを，以後 SSSP-MAXV と呼ぶ．

このプログラムでは，SSSP のためのレコード `SVal` (1 行目)，および MAXV のためのレコード `MVal` (3 行目) の二つをはじめに定義している．SSSP に関する初期化関数 `ssspinit` (6 行目) と LSS 関数 `ssspstep` (8 行目) は，図 1 と同じである．同様にして，MAXV に関する初期化関数 `maxvinit` (14 行目) とと LSS 関数 `maxvstep` (15 行目) も定義している．`maxvinit` においては，SSSP の結果の値を `val v .^ dist` によって取得し，それを初期値としている点に注意されたい．1 回目の `fregel` (21 行目) では入力グラフ `g` の SSSP を計算し，その結果として得られる最短路を各頂点に保持するグラフを `g1` としている．2 回目の `fregel` (22 行目) では，`g1` の MAXV を計算し，それを全体の結果としている (23 行目).

2.2　Fregel コンパイラ

図 3 に，Fregel コンパイラによる処理の流れを示す．はじめに正規化処理により，ユーザが記述した Fregel プログラムを正規化された Fregel プログラムに変換する．次にこの正規化された (関数型の) Fregel プログラムを，翻訳先のフレームワークが採用している Java，C++ などにより近い手続き的，かつ，フレームワーク独立な中間表現である FregelIR のコードへと変換する．最後に FregelIR のコードから，Giraph，Pregel+ などのフレームワークのプログラムを生成する．

3　正規化

正規化は，改良前の (Giraph プログラムだけを生成可能な) Fregel コンパイラでも行っており，その処理内容に特別な変更があったわけではないが，以降の処理の理解のために必要なので，ここで説明する．

SSSP-MAXV のプログラム例で示したように，Fregel のプログラムには `fregel` 等のグラフ高階関数を複数含むことがあり得る．それぞれのグラフ高階関数を独立な Pregel の計算によって実行させることも可能ではあるが，一般に Pregel 計算には大きな起動コストが必要である．我々は，複数のグラフ高階関数の使用を `fregel` をひとつだけ用いるプログラムに変換することにより，Pregel 計算の起動コストをなくし，効率化をはかっている．この変換を「正規化 (normalization)」と呼ぶ．SSSP-MAXV の場合，全体をまとめた単一の `fregel` は，はじめに SSSP の計算を行い，それが終わった後に引き続き MAXV の計算を行う．もとのプログラムにおける高階関数に対応する処理の段階を「フェーズ (phase)」と呼ぶ．すなわち，SSSP-MAXV の場合，SSSP (に含まれる `fregel`) は第 1 フェーズ，MAXV (に含まれる `fregel`) は第 2 フェーズの処理となる．

以下で，正規化の方針を SSSP-MAXV を例に用い

て説明する．

- 第 1 フェーズに引き続いて第 2 フェーズを行うために，SSSP と MAXV のためのレコード，初期化関数，LSS 関数を統合して，全体でひとつのレコード，初期化関数，LSS 関数を定義する．
- 全体の初期化関数は，最初に実行されるフェーズがどれかを指定する．
- 全体の LSS 関数は，ひとつの LSS において，いずれかのフェーズの初期化関数・LSS 関数・終了判定のいずれか (これをステップと呼ぶ) の処理を行う．
- 現在の処理がどのフェーズのどの処理であるかを指定するため，フェーズ番号とステップ番号の組を「状態」として持つ．全体の LSS 関数は，各フェーズ・ステップの計算処理の後，次の状態への状態遷移を明示的に行う．

このようにして正規化されたプログラムは `fregel` をひとつしか含まないような Fregel プログラムである．SSSP-MAXV の場合，その初期化関数は計算が第 1 フェーズから始まることを示し，LSS 関数は第 1 フェーズ (SSSP) の初期化関数・LSS 関数・終了判定および第 2 フェーズ (MAXV) の初期化関数・LSS 関数・終了判定を順次行うようなものである．このような LSS 関数は，状態を陽に扱い状態に応じて処理を分岐するような比較的大きな定義となり，(これを人間が読む必要はないが) 人間にとっては読みにくく理解しにくいものとなる．

4 中間表現 FregelIR

4.1 FregelIR の設計

我々が Fregel プログラムからの翻訳対象としている頂点主体グラフ計算フレームワークは，当面 Giraph [1]，Pregel+ [8]，GPS [6]，GraphX [4] としている．このうち GraphX はデータの表現および計算方法が Pregel と大きく異なるので FregelIR を通さずに翻訳することを考えている．それ以外のフレームワークについては，Giraph と GPS のプログラムは Java，Pregel+ のプログラムは C++ といった言語の違いはあるが，大雑把にいうと C 風のブロック構造，制御構造などを持つという点で共通点が多い．一方で，スーパーステップを記述する関数の定義方法，他頂点との間のメッセージ送受信の方法，集約の方法などは，フレームワークによって大きく異なる．また頂点を停止中に移行させるプリミティブ `voteToHalt()` の関数 (メソッド) 名など，細かな違いも多数ある．

以上より，ここで定義する中間表現 FregelIR は，フレームワークごとの違いを吸収できる程度の抽象度を持ちつつ，ブロック構造や制御構造 (条件分岐) などの共通部分に関しては，Java と C++ の最大公約数的な表現を持つように設計した．ここで注意すべきことは，FregelIR は Fregel プログラムを各フレームワークに翻訳できれば十分であるため，共通部分に関しては必要最低限の表現しか提供しない，という点である．たとえば，LSS 関数は，状態を保持しスーパーステップごとに状態遷移を行うような Java / C++ のメソッドに翻訳されるが，その内部の個別の計算において `while` 等の制御構造を用いるような反復計算は必要としない．そのため，FregelIR は反復のための制御構造に対応する表現を持っていない．

図 4 に，FregelIR の定義を一部簡略化したものを，Haskell の型定義として示す．

`IRProg` が，プログラム全体に対応する中間表現である．それは，各フェーズで使うデータ型，頂点，辺，メッセージ，集約子のデータ型の情報と，フェーズにおける計算処理の内容を表すデータ (`IRPhaseCompute` 型) から構成されている．

`IRPhaseCompute` 型は実質的には `IRPhaseComputeProcess` 型のデータのリストであり，ひとつの `IRPhaseComputeProcess` 型のデータが，ひとつの状態における計算処理を表している．ここには，状態，必要な局所変数の他，受信処理を含む計算本体のブロック，次の状態遷移のための条件と遷移先状態・送信処理を含むブロックが含まれる．

ブロックは，局所変数と文 (`IRStatement`) から成っている．文は，隣接頂点から値を得てまとめる操作，集約子への書き込み操作，出力辺に沿ったメッセージの送信など，フレームワークによる違いを吸収するのに十分な抽象度での定義になっている．

以下，SSSP において隣接頂点から値を得てまとめる操作を例として，この抽象度について説明する．

```
 1 data IRProg = IRProg String        -- プログラム名
 2               [IRTypeDecl]        -- 各フェーズが使うデータ型
 3               IRVertexStruct      -- 頂点のデータ型
 4               IREdgeStruct        -- 枝の値のデータ型
 5               IRMsgStruct         -- メッセージのデータ型
 6               IRAggStruct         -- 集約子のデータ型
 7               IRPhaseCompute      -- フェーズにおける計算処理
 8
 9 type IRNameAndType = (String, IRType)  -- 名前と型 (IRType の定義は省略)
10
11 data IRTypeDecl = IRTypeDecl String [IRNameAndType] -- 構造体名とメンバー
12
13 data IRVertexStruct = IRVertexStruct String          -- 名前
14                                      [IRNameAndType] -- メンバー: Prev/Curr 付きのもの
15                                      [IRNameAndType] -- メンバー: Prev/Curr なしでよいもの
16
17 data IREdgeStruct = IREdgeStruct String [IRNameAndType] -- 構造体名とメンバー
18
19 data IRMsgStruct = IRMsgStruct String [IRNameAndType] -- 構造体名とメンバー
20
21 data IRAggStruct = IRAggStruct String [(IRNameAndType, IRAggOp)] -- 構造体名と, 名前, 型, 用いる演算
22
23 type IRVertexComputeState = (Int, Int)     -- 状態 (フェーズ番号とステップ番号)
24
25 data IRPhaseCompute = IRPhaseCompute [IRPhaseComputeProcess]  -- 各 (phase,step) における処理内容
26
27 data IRPhaseComputeProcess =
28   IRPhaseComputeProcess IRVertexComputeState -- 状態
29                         [IRNameAndType]      -- 必要な局所変数
30                         IRBlock              -- 本体 (含受信処理)
31                         [(IRExpr, IRVertexComputeState, IRBlock)] -- 遷移条件, 次の状態, 通信
32
33 data IRBlock = IRBlock [IRNameAndType] [IRStatement] -- 局所変数と本体
34
35 data IRStatement =
36     IRStatementLocal IRVar IRExpr         -- ローカルな計算による値の変数への代入
37   | IRStatementMsg IRVar IRAggOp IRExpr   -- 隣接頂点の値の集約
38   | IRStatementReturn IRExpr              -- return 文
39   | IRStatementVTH                        -- voteToHalt
40   | IRStatementAggr IRNameAndType IRExpr  -- 集約子への書き込み
41   | IRStatementSendN IRNameAndType IRExpr -- (通常の辺に沿った) 送信メッセージのメンバーの名前/型と値
42
43 data IRVar = IRVarLocal IRNameAndType  -- 局所変数
44            | IRVarVertex IRNameAndType IRPrevCurr [IRNameAndType] -- 頂点のメンバー
45            | IRVarEdge IRNameAndType [IRNameAndType] -- 枝の値, フィールドアクセス
46            | IRVarAggr IRNameAndType                 -- 集約
47
48 data IRPrevCurr = IRPrev | IRCurr | IRNone
49
50 data IRExpr = IRIf IRExpr IRExpr IRExpr -- 条件分岐
51             | IRFunAp IRFun [IRExpr]    -- 関数/演算子適用
52             | IRVExp IRVar              -- 変数
53             | IRCExp IRType IRConst     -- 定数
54             | IRMVal IRNameAndType      -- 送信メッセージ用
55             | IRAggr IRNameAndType      -- 集約子からの集約結果値の獲得
56
57 data IRFun = IRFun String | IRBinOp String
58
59 data IRAggOp = IRAggMin | IRAggMax | IRAggSum | IRAggProd | IRAggAnd | IRAggOr
60
61 data IRConst = IRCInt Int | IRCBool Bool | IRCString String | IRCDouble Double
```

図 4 FregelIR の Haskell によるデータ型定義 (一部簡略化した)

SSSP の Fregel プログラムでは，内包表記を用いて隣接頂点から値を受け取り，辺の重みを加えて最小値をとっている (図 1: 9-10 行目)．この部分は，FregelIR では次のような IRStatement 型のデータとして表現される．

```
IRStatementMsg
  (IRVarLocal ("a", irInt))
  IRAggMin
  (IRMVal ("a", irInt) と辺の値の和)
```

ここで "a" はメッセージにつけられた固有の id で，結果の代入先の局所変数名も同じ "a" としている．また irInt は整数型を表す．翻訳先を Giraph とした時，次の段階の Giraph プログラムへの翻訳では，上は次のようになる．

```
int a = Integer.MAX_VALUE; // Min の単位元
for (IntWritable msg : messages)
  a = Math.min(a, msg.getValue() + 辺の値;
```

単位元の代入，メッセージに対する反復は，翻訳後において初めて出現し，FregelIR の段階ではこれらは IRStatementMsg という抽象度の高い構造の中に隠されている．

4.2 正規化された Fregel プログラムから FregelIR への変換

Fregel プログラムの翻訳処理は，Fregel プログラムの正規化の後，正規化された Fregel プログラムの FregelIR への変換，さらに FregelIR から各フレームワーク用のコード生成と進む．正規化により Fregel プログラムはひとつのグラフ高階関数からなるプログラムとなるが，中間表現 FregelIR との間には次のようなギャップがある．

- 関数型言語と手続き型言語
- LSS の (物理) スーパーステップへのマッピング
- 隣接頂点の参照を送信と受信とに分離

これらのギャップについて以下で説明する．

(正規化された) Fregel プログラムは Haskell サブセットの関数型プログラムである．一方，FregelIR は手続き型言語として設計した．特に，Haskell においてはプログラム中に循環を許すことが一番大きな問題となる．Fregel プログラムにおいて真に循環を含むようなプログラムは認めていないが，プログラムのそのままの順序で FregelIR に変換すると正しく動作しないことがある．また，Fregel における関数は，各計算の値を変数に束縛し，それらから一度にレコードを構築する．しかし，FregelIR (および生成されたコード) ではそのような一度の構築が不可能である場合もあり，レコードの構築をレコードのメンバへの代入として分割・適切に配置する必要もある．

LSS は (論理的な) 1 段階の処理を表現するが，隣接頂点の値を参照したり全体の値を集約したものを利用したりするには LSS を複数の (物理) スーパーステップへと分断する必要がある．すなわち，正規化された Fregel プログラムでは各フェーズは初期化・LSS・終了判定の 3 つのステップからなるが，FregelIR ではより多くのステップからなることがある．(物理) スーパーステップをまたいでローカル変数を保持することはできないので，必要があればそのような変数を頂点の値として定義しなければならない．

Fregel では隣接頂点の値を直接参照するようにプログラムを記述する．FregelIR (および生成されたコード) では，そのような処理は，明示的な送信と受信とに分ける必要がある．集約処理についても同様である．

5 フレームワークプログラムへの翻訳

5.1 概要

4.1 節で見たように，Fregel プログラム全体は IRProg 型で表現され，そこには各種データ型の情報，フェーズにおける計算処理の内容を表すデータが含まれている．これをもとに，次のようにフレームワークのプログラムを生成する．

まず各種データ型については，フレームワークごとに必要なデータ型をクラスあるいは構造体を用いて定義する．その際必要に応じてデータを追加することも行う．たとえば Pregel+ では，頂点の構造体の中に出力辺のベクタを保持する必要があるため，このような構造体メンバの追加などは 5.2 節で述べる方法を用いて個別に対処している．

フェーズにおける計算処理は，状態ごとに IRPhaseComputeProcess 型のデータが対応してい

る．そこで，各頂点で実行されるスーパーステップの関数 (メソッド) は，当該頂点から現在の状態を取得し，その状態に基づき switch 文を用いて処理を分岐させる．各 case では，その状態に対応する IRPhaseComputeProcess の処理を行うようにする．

5.2 型クラスの利用

フレームワーク独立の中間表現 FregelIR から各フレームワークの Java あるいは C++ コードへの翻訳は，Haskell により実装した．先に述べたように，中間表現からコードを生成する際，フレームワークによらない共通部分とフレームワーク依存部分がある．フレームワーク依存部分の生成には，Haskell の型クラスを利用した．

頂点を停止状態に移行させるコードの生成という簡単な例を用いて，フレームワーク依存のコード生成の基本的な考え方を説明する．停止状態への移行は，中間表現では IRStatement 型のデータ IRStatementVTH で表現されており，これに対応する実際のコードは，Giraph では当該頂点 vertex に対する vertex.voteToHalt() というメソッド呼出しであり，Pregel+ では vote_to_halt という関数呼出しである．

このようなコード生成を行うために，型クラス PregelGenerator を定義する (図 5)．この型クラスでは，フレームワーク依存のコード生成を行うためのいろいろな関数を定義する必要があり，その中のひとつに sVoteToHalt という voteToHalt() に対応する実際の関数 (メソッド) 呼出しを表す文字列を返す関数がある (5 行目)．

Giraph プログラム生成のために GiraphGenerator 型を定義 (9 行目) し，これを PregelGenerator のインスタンスとする．そのインスタンス宣言の中で，関数 sVoteToHalt を Giraph 用に定義 (14 行目) する．Pregel+ プログラム生成のためには PregelPlusGenerator 型を定義 (19 行目) し，インスタンス宣言を同様に記述する．

中間表現からフレームワークコードを生成するプログラム中の関数の大部分は，PregelGenerator クラスのインスタンス g を引数にとるように定義する．

```
1  -- 依存コード生成のための型クラス
2  class PregelGenerator g where
3    ...
4    -- VoteToHalt() に対応する文字列
5    sVoteToHalt :: g -> String
6    ...
7
8  -- Giraph プログラム生成用
9  data GiraphGenerator = GiraphGenerator
10
11 instance
12 PregelGenerator GiraphGenerator where
13   ...
14   sVoteToHalt GiraphGenerator =
15     "vertex.voteToHalt()"
16   ...
17
18 -- Pregel+プログラム生成用
19 data PregelPlusGenerator =
20   PregelPlusGenerator
21
22 instance
23 PregelGenerator PregelPlusGenerator where
24   ...
25   sVoteToHalt PregelPlusGenerator =
26     "vote_to_halt()"
27   ...
28
29 -- 中間表現 IRStatement の翻訳
30 ggStatement :: (PregelGenerator g) =>
31   g -> IRStatement -> [String]
32 ggStatement g IRStatementVTH =
33   [sVoteToHalt g]
34 -- 他のパターンマッチは省略
```

図 5 フレームワーク依存コードの生成

例として，中間表現 IRStatement を翻訳する関数 ggStatement の定義を，図 5 に示す．ggStatement の第 1 引数は g であり，ここには GiraphGenerator，PregelPlusGenerator など，翻訳先のフレームワークを表す PregelGenerator のインスタンスが与えられる．中間表現が IRStatementVTH の時には，その翻訳結果はフレームワーク依存となるので，sVoteToHalt g として g のインスタンス宣言の中で定義された関数が呼び出されるようにする．

このようにして，各フレームワークに対応する PregelGenerator のインスタンスを適切に定義すれば，依存部分はインスタンス宣言の中に閉じ込めることができ，コード生成部のモジュール性を高めることができる．

5.3 集約子への書き込みの翻訳

集約子への値の書き込み処理は，フレームワークに

より大きな違いがある．

Giraph の場合，スーパーステップのメソッドの中で

```
aggregate(name, value)
```

として，集約子 (名前で区別できる) `name` に値 `value` を書き込むことができる．

一方 Pregel+ では，スーパーステップの関数の中で `aggregate` のような関数を呼ぶのではなく，各スーパーステップが終わった後に集約を行うメソッド (具体的には `Aggregator` クラスの子クラスの `stepPartial`，`stepFinal`，`finishPartial`，`finishFinal`) が呼ばれ，実際の集約処理が行われるので，これらのメソッドを適切に定義しておかなければならない．

集約子への書き込みは，FregelIR では `IRStatemnet` 型に属すデータ `IRStatementAggr nt expr` で表現されている．ここで `nt` は集約子を表す名前とデータの型を表す組，`expr` は書き込む値を表す式である．

これに対する翻訳は，Giraph の場合 `expr` を翻訳してから単純に `aggregate` を呼ぶようにするだけで良いが，Pregel+ の場合はスーパーステップ終了後の集約処理で正しく集約が行われるようにする必要がある．そのため，集約子をまとめた構造体 `AggData` を定義し，さらに頂点の構造体の中に，以下のメンバを持つようにした．

- スーパーステップ後の集約処理が必要であることを表すメンバ `needAgg`．ここには，書き込む集約子の番号を保持する．
- 集約子に書き込む値を保持する `AggData` 型のメンバ `aggv`

そうすると，`IRStatementAggr nt expr` に対する翻訳は，当該頂点の構造体中の `needAgg` に集約子の番号を代入し，`aggv` に `expr` の計算結果を代入すれば良い．さらに `stepPartial` メソッドは，`needAgg` を参照して適切な集約が行われるような処理を行うように Pregel+ のコードを出力する．

6 おわりに

本稿では，Fregel を複数の頂点主体グラフ計算フレームワークに対応させるために設計した，フレームワーク独立な中間表現 FregelIR の設計と，正規化された Fregel プログラムから FereglIR への変換と FregelIR からのコード生成の方法を述べた．現状では，小さな例題を用いた動作の検証しか行われていない．今後は，複数の例題を用いてコード生成するとともに，Giraph と Pregel+ 以外のフレームワークへの翻訳も実現し，本稿で述べた中間表現とコード生成の実用性を検証していく必要がある．また，中間表現上において，または，コード生成時に最適化を行うことにより，生成されたコードの性能を吟味することも必要である．

謝辞 本研究の一部は，JSPS 科研費 JP26280020，JP15K15974 の助成を受けたものである．

参 考 文 献

[1] Apache Giraph: The Giraph Website. `http://giraph.apache.org/`.
[2] Coll Ruiz, O., Matsuzaki, K., and Sato, S.: S6Raph: Vertex-centric Graph Processing Framework with Functional Interface, *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, FHPC 2016, New York, NY, USA, ACM, 2016, pp. 58–64.
[3] Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A., and Iwasaki, H.: Think Like a Vertex, Behave Like a Function! A Functional DSL for Vertex-centric Big Graph Processing, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, ACM, 2016, pp. 200–213.
[4] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework, *11th USENIX Symposium on Operating Systems Design and Implement ation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, Flinn, J. and Levy, H.(eds.), USENIX Association, 2014, pp. 599–613.
[5] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: A System for Large-scale Graph Processing, *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, ACM, 2010, pp. 135–146.
[6] Salihoglu, S. and Widom, J.: GPS: A Graph Processing System, *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, ACM, 2013, pp. 22:1–22:12.
[7] Valiant, L. G.: A Bridging Model for Parallel Computation, *Communications of the ACM*, Vol. 33, No. 8(1990), pp. 103–111.
[8] Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., and

Bu, Y.: Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees, *Proceedings of the VLDB Endowment*, Vol. 7, No. 14(2014), pp. 1821–1832.

[9] 江本健斗, 松崎公紀, 胡進行, 森畑明昌, 岩崎英哉: 大規模グラフ並列処理のための関数型領域特化言語 Fregel とその評価, 日本ソフトウェア科学会第 *33* 回大会 *(2016* 年度*)* 講演論文集.