グラフの探索関数の再帰的定義と変換

篠埜 功　胡 振江　武市 正人

グラフアルゴリズムを再帰的に記述し，プログラム変換を行おうとする試みがこれまでにいくつかなされているが，それらは深さ優先探索を対象としており，提案されている変換規則も複雑である．本論文では，深さ優先探索をその特別の場合として含むグラフの一般的探索を行う関数を再帰的に定義する．この関数を hylomorphism と呼ばれる形に変換することにより，融合変換やその他の各種の変換を行うことができ，効率的なアルゴリズムを導出することができる．本稿ではさらに，グラフの一般的探索関数を関数型言語 Haskell によって実現する方法を示す．

1 はじめに

プログラムの信頼性，生産性の向上のためにアルゴリズムを再帰的に記述し，正しさの証明，プログラム変換による効率改善，仕様からのプログラム導出などを行う手法に関する研究が以前から行われている．リスト，木の上のアルゴリズムに関しては多くの研究がなされており [2]，最近ではグラフアルゴリズムに関しても再帰的記述がいくつか試みられている [5] [6] [8]．しかし，グラフアルゴリズムの再帰的記述は現在のところ，深さ優先探索のみを対象としたもの [8]や，有向無閉路グラフのみを対象としたもの [6]など，適用範囲を限定して行われている．また，[5]においては深さ優先探索をする関数に関してプログラム変換をするための融合規則も提案されているが，非常に複雑であり，規則の適用が難しく，実用上の問題点がある．

本論文では，グラフの一般的探索 [18]を行う関数を再帰的に記述する．これは深さ優先探索，幅優先探索をその特別の場合として含み，適用範囲が広いものである．この一般的探索関数は hylomorphism という形に変換することができ [7] [16]，それにより，hylo fusion と呼ばれる融合変換やその他いろいろな変換を行うことができる．本稿ではさらに，グラフの一般的探索関数を関数型言語 Haskell によって効率よく実現する方法を示す．

本論文の構成は以下のようになっている．まず，2 節で関連研究について述べる．次に 3 節でグラフの一般的探索関数を再帰的に記述する．4 節で一般的探索関数を hylomorphism という形に変換し，例として融合変換を行う．5 節で一般的探索関数の効率のよい実現法を示す．6 節で結論を述べる．

2 関連研究

[8] [9]においては，深さ優先探索木を生成する関数を state transformer [10] [12]の技法を用いて関数型言語で記述している．この関数と他の関数を組み合わせることにより，深さ優先探索に関するさまざまなアルゴリズムを記述している．これにより，プログラムの正当性の証明が行いやすくなっており，グラフの強連結成分分解を行うプログラムの正当性の証明を行っている． [9]においてはプログラムの融合変換を行っているが，state transformer を用いている部分については融合変換が複雑になる．また，融合変換は深さ優先探索にのみ限定され

A General Recursive Form for Graph Traversals and its Transformation.

Isao Sasano, Zhenjiang Hu, Masato Takeichi, 東京大学大学院工学系研究科情報工学専攻, Department of Information Engineering, University of Tokyo.

コンピュータソフトウェア, Vol.17, No.3 (2000), pp.2–19.
[論文] 1999 年 5 月 31 日受付.

る.

[5]においては, アクティブパターンマッチングと呼ばれる技法を用いることにより, グラフを再帰的に取り扱っており, その上で深さ優先探索を関数型言語を用いて再帰的に記述している. また, 深さ優先探索関数に関する融合規則が述べられており, 融合変換が行われているが, 融合規則が非常に複雑であり, 規則の適用が難しく, 実用上の問題点がある.

[6] [14]においては, グラフアルゴリズムを代数的枠組で取り扱っている. [6]においては, 対象を有向無閉路グラフに限定し, 実現に関しては述べられていない. [14]においては, 関数ではなく, 関係を用いてアルゴリズムを記述しており, 現段階においては実用的ではない.

[17]においては, グラフ上のある帰納的な関数に関する融合規則を提示し, それを用いて最短路問題を解くダイクストラ法 [3]を導出しているが, 深さ優先探索, 幅優先探索は扱っていない.

[4]においては, 本論文で取り扱うグラフの一般的探索を取り扱っているが, 再帰的には記述されておらず, またプログラム変換は行われていない.

3 グラフの一般的探索関数の定義

この節では, 有向グラフを再帰的に定義し, その上で一般的探索関数を再帰的に記述する. 本論文で取り扱うグラフは有向グラフとする. これまでは, グラフアルゴリズムを再帰的に取り扱うときには, 深さ優先探索などに範囲を限定していたが, 本論文で述べる一般的探索は, 深さ優先探索, 幅優先探索をその特別の場合として含むものであり, 適用範囲の広いものである.

3.1 グラフの再帰的定義

グラフを表現するときには, 行列や隣接リストなどを用いるのが一般的であるが, 次のように再帰的に定義することもできる [5].

$$\begin{array}{rcl} \mathit{Graph} & ::= & \mathit{Empty} \\ & \mid & \mathit{Context}\ \&\ \mathit{Graph} \\ \mathit{Context} & ::= & ([\mathit{Vertex}], \mathit{Vertex}, [\mathit{Vertex}]) \end{array}$$

グラフは, 空のグラフ Empty であるか, または, グラフに $\mathit{Context}$ 型の要素 1 つを構成子 & によって付け加えたものとして定義する. $\mathit{Context}$ 型の要素は 1 つの頂点に関する情報を表しており, その第 1 要素はその頂点を終点とする枝の始点のリスト, 第 2 要素はその頂点自身, 第 3 要素はその頂点を始点とする枝の終点のリストを表す. Vertex は頂点の型を表す. 例えば, 図 1のグラフは

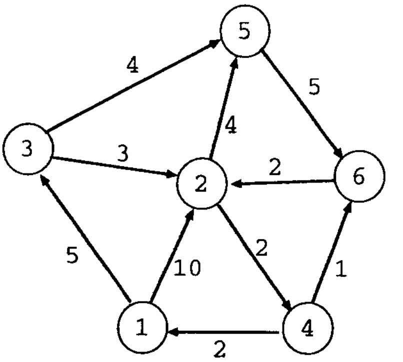

図 1 有向グラフの例

$$\begin{array}{rcl} ([4],1,[2,3]) & \& & ([3,6],2,[4,5]) \\ & \& & ([\,],3,[5]) \\ & \& & ([\,],4,[6]) \\ & \& & ([\,],5,[6]) \\ & \& & ([\,],6,[\,]) \\ & \& & \mathit{Empty} \end{array}$$

のように表現できる. ただし & は右結合的であるとし, 枝の重みは枝を引数にとる関数weightで与えられることにする. 図 1のグラフでは, 関数weightは,

$$\begin{array}{rcl} \mathit{weight}\ (1,2) & = & 10 \\ \mathit{weight}\ (1,3) & = & 5 \\ \mathit{weight}\ (2,4) & = & 2 \\ \mathit{weight}\ (2,5) & = & 4 \\ \mathit{weight}\ (3,2) & = & 3 \\ \mathit{weight}\ (3,5) & = & 4 \\ \mathit{weight}\ (4,1) & = & 2 \\ \mathit{weight}\ (4,6) & = & 1 \\ \mathit{weight}\ (5,6) & = & 5 \\ \mathit{weight}\ (6,2) & = & 2 \end{array}$$

と定義される.

3.2 グラフの一般的探索関数の再帰的記述

グラフの探索関数は, 頂点を 1 つずつ渡りながらある計算を行うという形で自然に記述することができる. 例

えば，グラフを引数にとり，結果として頂点数を返す関数 $nodeNumber$ を定義することを考えてみる．

$$nodeNumber : Graph \rightarrow Int$$

これには，引数に与えられたグラフの頂点を 1 つずつ渡りながら，頂点を 1 つ渡るごとに値を 1 増加させるという計算を行えばよいので，

$$nodeNumber\ Empty = 0$$

$$nodeNumber\ ((p, v, s)\ \&\ g) = 1 + nodeNumber\ g$$

のように定義すればよい．これは，グラフが $Empty$ の場合には 0 を返し，$Empty$ でない場合には頂点 v とそれに付随する枝を除いた残りのグラフ中の頂点数に 1 を加えたものを結果として返すことを意味している．

この例のように，頂点の訪問順序が自由でよい場合には単純に記述できるが，深さ優先探索，幅優先探索，これから記述する一般的探索などのように，次に訪問する頂点を指定したい場合には，引数に与えられるグラフを，次に訪問する頂点 (とそれに付随する枝) と残りのグラフとに分割するということが必要になる．このような分割を記述するために，アクティブパターンマッチング [5]と呼ばれる技法を用いる．この技法を用いると，例えば，グラフ g において，ある頂点 v から出ている枝の先の頂点 (successor) を求める関数は，

$$succ : Vertex \rightarrow Graph \rightarrow [\,Vertex\,]$$

$$succ\ v\ ((p, v, s)\ \underline{\&}\ g) = s$$

のように記述することができる．アクティブパターンマッチングを明示的に表すときは，$\underline{\&}$ のように & に下線をつけて表すものとする．

このアクティブパターンマッチングと呼ばれる技法を用いることにより，自然に一般的探索関数を再帰的に記述することができる．グラフの一般的探索 [18]とは，深さ優先探索，幅優先探索と同様に，グラフ中の頂点，枝を何らかの順番で訪問していくことである．その際の訪問順序の決め方の基準は探索の深さには限定しない．その基準として，出発点からの距離をとれば最短路問題を解くダイクストラ法になり，枝の重みにすれば最小木問題を解く Prim のアルゴリズム [13]になる．探索の様子は図 2のようになる．　黒く塗りつぶした頂点が訪問済みの頂点を表し，斜線のついた頂点が次に訪問する頂点の候補 (前線と呼ぶ [18]) を表している．具体的には，前線は，その頂点を訪問した直前の頂点，前線中の頂点自身，評価値の 3 つ組を要素とする集合であるとし，前線の型名は $Frontier$ とする．評価値の型は $Value$ とする．前線に対する操作は，要素の追加，評価値が最小の要素 (のうちの任意の 1 つ) の取りだしの 2 つのみとする．要素の追加は，演算子 $\uplus$ によって行い，評価値最小要素の取りだしは $\triangleleft$ によって行う．$(v, x, d) \uplus fr$ は前線 fr に要素 (v, x, d) を追加してできる前線を表し，$(pv, v, d) \triangleleft fr$ は，評価値最小の要素が (pv, v, d)，頂点 v を第 2 要素に持つ要素をすべて除いた前線が fr であるような前線を表す．空の前線は$EmpFrontier$で表す．次に訪問する頂点は，前線中の評価値最小のものとすればよいので，前線を，$(pv, v, d) \triangleleft fs$ によって評価値最小要素 (pv, v, d) とそれを除いた前線 fs に分解し，頂点 v を次に訪問する頂点とすればよい．

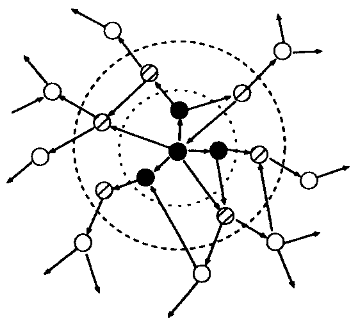

図 2　グラフの探索

頂点を 1 つ訪問するごとに前線は更新されるが，これを行う関数を

$$\begin{aligned} merge : (\,&Vertex \rightarrow Value \rightarrow Vertex \rightarrow Value) \\ &\rightarrow Vertex \rightarrow Value \rightarrow [\,Vertex\,] \\ &\rightarrow Frontier \rightarrow Frontier \end{aligned}$$

$$merge\ eval\ v\ d\ [\,]\ fr = fr$$

$$\begin{aligned} &merge\ eval\ v\ d\ (x : s)\ fr \\ &\quad = \mathrm{let}\ d' = eval\ v\ d\ x \\ &\qquad \mathrm{in}\ merge\ eval\ v\ d\ s\ ((v, x, d') \uplus fr) \end{aligned}$$

と定義する．第 1 引数には前線に新たに挿入する要素の評価値を計算する関数，第 2 引数には訪問中の頂点，第 3 引数には訪問中の頂点の評価値，第 4 引数には訪問中の頂点の，未訪問の successor のリスト，第 5 引数には更新前の前線が与えられる．

この関数 $merge$ を用いることにより，一般的探索を行う関数は，前線とグラフの対の上で再帰的に記述するこ

とができ，

$$f : (\mathit{Frontier}, \mathit{Graph}) \to A$$
$$f\ (fr, \mathit{Empty}) = e$$
$$f\ ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g)$$
$$= acc\ ((pv, v, d),\ f\ (merge\ eval\ v\ d\ s\ fr,\ g))$$

のように書き表すことができる．ただし，この関数 f は，$e, acc, eval$ に依存しており，

- e はグラフが Empty の場合の値，
- acc は非再帰部分と再帰部分の値からある計算を行う関数，
- $eval$ は前線に新たに追加する要素の評価値を計算する関数

を表している．$eval$ によって頂点の訪問順序が決まり，e, acc によって，どのような計算が行われるかが決まる．$e, acc, eval$ を適切に指定することによって，さまざまなアルゴリズムを表現することができる (3.3節)．

なお，ここで定義した f は，引数にとるグラフ中に，初期前線から到達不可能な頂点がある場合にはマッチする場合がなくなってしまうが，分かりやすくするため，また，後で述べる融合変換を行いやすくするためにこのような定義を用いることとし，引数としてとるグラフは，初期前線からすべての頂点が到達可能であると仮定する．

以上で定義した関数 f を，$e, acc, eval$ を引数にとる関数として記述することとし，この関数を $explore$ と名付ける．

$$f = explore\ acc\ e\ eval$$

例として，グラフの頂点数を求める関数 $nodeNumber$ を $explore$ を用いて記述すると，

$$nodeNumber : \mathit{Graph} \to \mathit{Int}$$
$$nodeNumber\ g = explore\ acc\ e\ eval\ (fr, g)$$
$$\text{where } acc\ (x, y) = 1 + y$$
$$e = 0$$

のようになる．関数 $eval$，初期前線 fr は適当に与えればよい．

3.3 グラフの一般的探索関数の例

以上で定義した一般的探索関数は深さ優先探索，幅優先探索を含む探索一般を記述しうるものであり，記述力が高い関数である．以下では一般的探索関数の記述力の高さを示すために，ダイクストラ法で最短距離を求める関数，および深さ優先探索木を求める関数を記述する．

3.3.1 例 1— ダイクストラ法

グラフの 2 点間の最短距離を求めるアルゴリズムの 1 つにダイクストラ法 [3]がある．ダイクストラ法は，深さ優先探索，幅優先探索では書き表すことはできないが，3.2節で記述した関数 f のような形を用いると，自然に再帰的に記述することができる．ある 1 つの頂点 v とグラフ g を引数にとり，頂点 v からグラフ g 中の他のすべての頂点への最短距離を求める関数 $dijkstra$ を自然に再帰的に記述すると次のようになる．

$$dijkstra : \mathit{Vertex} \to \mathit{Graph} \to [(\mathit{Vertex}, \mathit{Value})]$$
$$dijkstra\ v\ g = dijkstra'\ (fr, g)$$
$$\text{where } fr = toFrontier\ [(\_, v, 0)]$$
$$dijkstra' : (\mathit{Frontier}, \mathit{Graph}) \to [(\mathit{Vertex}, \mathit{Value})]$$
$$dijkstra'\ (fr, \mathit{Empty}) = [\,]$$
$$dijkstra'\ ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g)$$
$$= (v, d)\ :\ dijkstra'\ (merge\ eval\ v\ d\ s\ fr, g)$$
$$\text{where}$$
$$eval\ v\ d\ x = d + weight\ (v, x)$$

$toFrontier$ は，引数に与えられたリスト中のすべての要素を，空の前線に追加してできる前線を返す関数であるとする．

$$toFrontier\ :\ [(\mathit{Vertex}, \mathit{Vertex}, \mathit{Value})] \to \mathit{Frontier}$$
$$toFrontier\ [\,] = EmpFrontier$$
$$toFrontier\ (x : xs) = x \uplus (toFrontier\ xs)$$

また，初期前線 fr の定義式の右辺の _ は，仮想的な頂点 (引数に与えられるグラフ中のどの頂点とも異なる頂点) を表す．この関数 $dijkstra$ は 3.2節で定義した関数 $explore$ を用いて記述すると，

$$dijkstra : \mathit{Vertex} \to \mathit{Graph} \to [(\mathit{Vertex}, \mathit{Value})]$$
$$dijkstra\ v\ g = explore\ acc\ e\ eval\ (fr, g)$$
$$\text{where}$$
$$acc\ ((pv, v, d), r) = (v, d)\ :\ r$$
$$e = [\,]$$
$$eval\ v\ d\ x = d + weight\ (v, x)$$
$$fr = toFrontier\ [(\_, v, 0)]$$

のようになる．この関数$dijkstra$に頂点 v, グラフ g を引数として与えると, 頂点, その頂点への頂点 v からの最短距離の対のリストが結果として得られる．例えば, 図 1のグラフ (これを ex とする) において頂点 1 から他の頂点への最短距離を求めるには, $dijkstra\ 1\ ex$ を計算すればよく, 結果は,

$$[(1,0),(3,5),(2,8),(5,9),(4,10),(6,11)]$$

となる．

3.3.2 例 2— 深さ優先探索

深さ優先探索は重要な探索法であり, 深さ優先探索を用いることにより, さまざまなグラフアルゴリズムを記述することができる．しかし, 深さ優先探索は一般的探索の 1 つの例にしかすぎないので, 深さ優先探索関数は3.2節で定義した一般的探索関数$explore$を用いて記述することができる．ここでは, 深さ優先探索関数の一例である, 深さ優先探索木を求める関数dfsを一般的探索関数$explore$を用いて記述する．深さ優先探索は一般的探索において出発点からの深さを前線の評価基準として用いることにより表すことができるので, 深さ優先探索木を返す関数dfsは,

$$
\begin{aligned}
& dfs : [\mathit{Vertex}] \rightarrow \mathit{Graph} \rightarrow \mathit{Tree\ Vertex}\\
& dfs\ vs\ g = dfs'\ (fr, g)\ (\mathit{Node}\ \_\ [\,])\\
& \quad \text{where}\\
& \qquad dfs'\ (fr, \mathit{Empty})\ r = r\\
& \qquad dfs'\ ((pv,v,d) \triangleleft fr,\ (p,v,s)\ \underline{\&}\ g)\ r\\
& \qquad\quad = dfs'\ (merge\ eval\ v\ d\ s\ fr, g)\\
& \qquad\qquad\quad (addv\ (pv,v)\ r)\\
& \qquad eval\ v\ d\ x = d-1\\
& \qquad fr = toFrontier\\
& \qquad\qquad [(\_,v,i) | (v,i) \leftarrow (zip\ vs\ [1..\#vs])]
\end{aligned}
$$

のように表すことができる．$Tree$ は

$$\mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a]$$

と定義される．$\#vs$ はリスト vs の長さを表し, $[1..n]$ は $[1,2,\ldots,n]$ というリストを表す．zip は 2 つのリストを引数にとり, 対応する要素の対のリストを返す関数を表す．また, dfs' の型は,

$$
\begin{aligned}
dfs' : (\mathit{Frontier}, \mathit{Graph}) & \rightarrow \mathit{Tree\ Vertex}\\
& \rightarrow \mathit{Tree\ Vertex}
\end{aligned}
$$

である．この関数 dfs は, 仮想的な頂点 $\_$(引数に与えられるグラフ中のどの頂点とも異なる頂点) とその頂点から vs 中の頂点への枝をグラフ g に加え, 頂点 $\_$ を出発点として深さ優先探索を行い, 仮想的な頂点 $\_$ を根とする探索木を結果として返すことを表している．新たに前線に加える要素の評価値は訪問中の頂点の評価値 d から 1 を引いた値 $d-1$ にしているが, これは, 探索の出発点からの深さが深い頂点ほど評価値が小さく, すなわち優先度が高くなることを表しており, これを $d+1$ にすると, 関数 dfs は幅優先探索木を返す関数を表すようになる．$addv$は

$$
\begin{aligned}
& addv : (\mathit{Vertex}, \mathit{Vertex}) \rightarrow \mathit{Tree\ Vertex}\\
& \qquad\qquad\qquad\qquad\qquad \rightarrow \mathit{Tree\ Vertex}\\
& addv\ (pv,v)\ (\mathit{Node}\ v'\ cs) =\\
& \quad \text{if } (pv == v')\\
& \quad \text{then } (\mathit{Node}\ v'\ (cs\ +\!\!+\ [\mathit{Node}\ v\ [\,]]))\\
& \quad \text{else } \mathit{Node}\ v'\ [addv\ (pv,v)\ c \mid c \leftarrow cs]
\end{aligned}
$$

という関数であり, 第 1 引数に与えられる枝 (pv, v) の終点 v を第 2 引数に与えられる木 r 中の頂点 pv の子として付け加えることを意味している．なお, 初期前線 fr の評価値を関数 dfs の第 1 引数に与えられる頂点リスト中の要素の先頭から順に $1,2,\ldots$ としているのは, 関数 dfs の第 1 引数に与えられる頂点リスト中の要素の順番に意味を持たせるためである．4.3.2節で記述する強連結成分分解を行う関数等においては, この順番が意味を持つ．関数dfsは 3.2節で定義した関数$explore$を用いて記述すると

$$
\begin{aligned}
& dfs\ vs\ g = explore\ acc\ e\ eval\ (fr, g)\ (\mathit{Node}\ \_\ [\,])\\
& \quad \text{where}\\
& \qquad acc\ ((pv,v,d), r) = r\ \circ\ addv\ (pv,v)\\
& \qquad e = id\\
& \qquad eval\ v\ d\ x = d-1\\
& \qquad fr = toFrontier\\
& \qquad\qquad [(\_,v,i) | (v,i) \leftarrow (zip\ vs\ [1..\#vs])]
\end{aligned}
$$

と書き表すことができる．

4 一般的探索関数の変換

前節で定義した一般的探索関数と他の関数とを組み合わせることにより, さまざまなアルゴリズムを表現することができる．しかし, そのようにして書かれたプログラムは効率が悪いことがあり, 効率を改善するために, プログラム変換が行われる．今までにグラフ上の関数につ

いて行われていた変換は，独自に定義した探索関数のみに限定された変換であったが [5] [9]，本論文では，前節で定義した一般的探索関数を hylomorphism [7] [16]と呼ばれる形に記述し直してから変換を行う．これは本論文の重要な貢献の1つである．これにより，すでによく知られている枠組でさまざまな変換を行うことができる．ここでは，例として，重要な変換である融合変換を行う．4.1で，hylomorphism の基本概念，表記法について簡潔に説明したのち，4.2で一般的探索関数 *explore* を hylomorphism で記述し，4.3で，融合変換を行った例を示す．

4.1 Hylomorphism の定義

この節では，カテゴリー (category) の理論に現れる，関手 (functor) について説明を行ったのち，hylomorphism の定義を行う．以下で述べることは，カテゴリー理論で知られていることであり，[7] [16]等にも述べられている．以下では，カテゴリーは CPO (すべての完備半順序集合 (complete partially ordered set) を対象 (object) とし，その間の連続関数 (continuous function) を射 (arrow) とするカテゴリー) であるとし，関手は，CPO から CPO への関手のみを考える．

4.1.1 関手

関手は型を引数にとると型を返し，関数を引数にとると関数を返す．4.2節において，一般的探索関数 *explore* の hylomorphism による記述を行うが，そこに現れる関手は以下の4つの基本的な関手のみを用いて構成することができる．以下の4つの関手から構成される関手は，polynomial functor と呼ばれる．

定義 1 (identity functor)
identity functor I の，型 X，関数 f に対する操作は以下のように定義される．

$$\begin{aligned} I\ X &= X \\ I\ f &= f \end{aligned}$$

□

定義 2 (constant functor)
constant functor $!A$ の，型 X，関数 f に対する操作は以下のように定義される．

$$\begin{aligned} !A\ X &= A \\ !A\ f &= id \end{aligned}$$

ここで，A はある型を表し，id は恒等関数を表す． □

定義 3 (product functor)
product functor $\times$ の，型 X, Y，関数 f, g に対する操作は以下のように定義される．

$$\begin{aligned} X \times Y &= \{(x, y) \mid x \in X, y \in Y\} \\ (f \times g)\ (x, y) &= (f\ x, g\ y) \end{aligned}$$

なお，$\times$ は，関手間の次のような二項演算子としても使う．

$$\begin{aligned} (\mathsf{F} \times \mathsf{G})\ X &= \mathsf{F}\ X \times \mathsf{G}\ X \\ (\mathsf{F} \times \mathsf{G})\ p &= \mathsf{F}\ p \times \mathsf{G}\ p \end{aligned}$$

□

定義 4 (coproduct functor)
coproduct functor $+$ の，型 X, Y，関数 f, g に対する操作は以下のように定義される．

$$\begin{aligned} X + Y &= \{1\} \times X \cup \{2\} \times Y \\ (f + g)\ (1, x) &= (1, f\ x) \\ (f + g)\ (2, x) &= (2, g\ x) \end{aligned}$$

第1式の右辺の $\times$ は，直積集合を求める演算子であり，1, 2 はタグを表している．タグに関連する演算子 $\triangledown$ をここで定義しておく．

$$\begin{aligned} (f \triangledown g)\ (1, x) &= f\ x \\ (f \triangledown g)\ (2, x) &= g\ x \end{aligned}$$

なお，$+$ は，関手間の次のような二項演算子としても使う．

$$\begin{aligned} (\mathsf{F} + \mathsf{G})\ X &= \mathsf{F}\ X + \mathsf{G}\ X \\ (\mathsf{F} + \mathsf{G})\ p &= \mathsf{F}\ p + \mathsf{G}\ p \end{aligned}$$

□

4.1.2 関手とデータ型との対応について

関手が polynomial functor であるときには，その関手はあるデータ型を表しているとみなすことができる．また，データ型から，それを表す関手を導くこともできる [15]．ここでは，リストと木を例として，関手とデータ型との対応を示す．

- リスト

  型 A の要素を持つリストは，

  $$List\ A = Nil \mid Cons\ (A, List\ A)$$

  のように定義することができる．この型 $List\ A$ に対応する関手は，

  $$F_L = !\mathbf{1}\ +\ !A \times I$$

  である．この関手 F_L は，$(List\ A)$ 型を定める．**1**

は，要素を 1 つだけ持つ集合を表しており，**1** の要素は () であるとする．次の関数 in_{F_L} は $List\ A$ のデータ構成子を表す．

$$\begin{aligned} in_{F_L} &: F_L(List\ A) \rightarrow (List\ A) \\ in_{F_L} &= (\lambda().Nil) \triangledown Cons \end{aligned}$$

$\lambda().Nil$ は () を引数にとり Nil を返す定数値関数を表しているが，式が繁雑になるのを避けるため，これを単に Nil と記述することもある．

- 木

  型 A の要素を持つ 2 分木は

  $$Tree\ A = Leaf\ A \mid Node\ (Tree\ A, Tree\ A)$$

  のように定義することができる．この型 $Tree\ A$ に対応する関手は

  $$F_T = \ !A\ +\ I \times I$$

  である．この関手 F_T は，$(Tree\ A)$ 型を定める．次の関数 in_{F_T} は $Tree\ A$ のデータ構成子を表す．

  $$\begin{aligned} in_{F_T} &: F_{T_A}(Tree\ A) \rightarrow (Tree\ A) \\ in_{F_T} &= Leaf \triangledown Node \end{aligned}$$

4.1.3 hylomorphism の定義

3 つ組からなる hylomorphism は以下のように定義される [16].

定義 5 (hylomorphism)

F, G を関手 (functor) とし，関数 $\phi : GA \rightarrow A$, $\psi : B \rightarrow FB$ と自然変換 (natural transformation) $\eta : F \dot{\rightarrow} G$ が与えられたとき，hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : B \rightarrow A$ は次の等式を満たす最小不動点として定義される．

$$f = \phi \circ \eta \circ F\,f \circ \psi$$

$[\![\phi, \eta, \psi]\!]_{G,F}$ の添字の G, F は，自明なときは省略する． □

関数 ϕ, η, ψ はそれぞれ，引数の再帰データの分解，分解された各要素に対する処理，結果を表す再帰データの組み立て，の役割を果たす．hylomorphism の記述力は高く，ほとんどの再帰的な関数は hylomorphism で記述することができ，また，hylomorphism に関する変換規則として，hylo shift 定理，hylo fusion 定理，酸性雨定理などが知られており，それを利用することにより，さまざまな変換を行うことができる [7] [16]. hylomorphism は，以下に述べる，catamorphism, anamorphism という有用な再帰形式をその特別の場合として含んでおり，非常に一般的な再帰形式である．

定義 6 (catamorphism, anamorphism)

T_F を，関手 F によって定まるデータ型とする．

$$\begin{aligned} (\![\_]\!)_F &: \forall A.\ (F\ A \rightarrow A) \rightarrow T_F \rightarrow A \\ (\![\phi]\!)_F &= [\![\phi, id, out_F]\!]_{F,F} \end{aligned}$$

$$\begin{aligned} [\!(\_)\!]_F &: \forall A.\ (A \rightarrow F\ A) \rightarrow A \rightarrow T_F \\ [\!(\psi)\!]_F &= [\![in_F, id, \psi]\!]_{F,F} \end{aligned}$$

□

catamorphism $(\![\_]\!)$ は，リストを引数にとる関数 foldr の一般化になっており，anamorphism $[\!(\_)\!]$ は，その双対である．hylomorphism $[\![\phi, id, \psi]\!]_{F,F} : B \rightarrow A$ は catamorphism $(\![\phi]\!)_F : T_F \rightarrow A$ と anamorphism $[\!(\psi)\!]_F : B \rightarrow T_F$ の合成関数に分解することができ，

$$[\![\phi, id, \psi]\!]_{F,F} = (\![\phi]\!)_F \circ [\!(\psi)\!]_F$$

が成り立つ．これは，$[\!(\psi)\!]_F$ によって T_F 型の中間データが生成され，それが $(\![\phi]\!)_F$ によって消費されるということを表している．hylomorphism $[\![\phi, id, \psi]\!]_{F,F}$ はこの 2 つの関数 $(\![\phi]\!)_F$, $[\!(\psi)\!]_F$ が融合されたものを表している．

4.1.4 hylomorphism で表現してよい条件

リスト，木などの，自由 (free) な再帰的データ型に関しては，hylomorphism の理論は適用可能であるが，本論文で採用したグラフの定義では，1 つのグラフを表すのに複数の表現が可能であり，このような，自由な再帰的データ型でないデータ型に関して hylomorphism の理論が適用できるかどうかは自明ではないように見える．そこで，hylomorphism で表現してよい条件を以下で明確にしておく．

$[\![\phi, \eta, \psi]\!]_{G,F} : B \rightarrow A$ と記述してよい条件としては，

- F-algebra のカテゴリー，G-algebra のカテゴリーに始対象が存在する
- ϕ の型が $GA \rightarrow A$ である
- ψ の型が $B \rightarrow FB$ である
- η が $F \dot{\rightarrow} G$ 型の自然変換である

が挙げられ，これらを満たしていれば十分である．A, B は必ずしも自由な再帰的データ型である必要はない．4.2節で述べるが，関数 *explore* の hylomorphism 表現における関手は

$$F = \ !\mathbf{1}\ +\ (!(Vertex, Vertex, Value) \times I)$$

であり，これは $(Vertex, Vertex, Value)$ 型の要素を持つリスト型を定義する関手である．F-algebra のカテゴリーには始対象が存在しており，その他の条件は満たされているので，4.2節で記述する，関数 *explore* の hylomorphism 表現は妥当であるといえる．一旦 hylomorphism で表現されると，あとは hylomorphism の変換規則によって自由に変換を行うことができる．

4.2 グラフの一般的探索関数の hylomorphism による記述

3.2節で定義した一般的探索関数 *explore* を hylomorphism の形で記述すると，次のようになる．

$$
\begin{aligned}
&explore\ acc\ e\ eval = [\![e \triangledown acc,\ id,\ \psi]\!]_{F,F} \\
&\quad \text{where} \\
&\qquad \psi\ (fr, Empty) = (\mathbf{1},\ ()) \\
&\qquad \psi\ ((pv, v, d) \triangleleft fr,\ (p, v, s)\ \underline{\&}\ g) \\
&\qquad\quad = (2,\ ((pv, v, d), (merge\ eval\ v\ d\ s\ fr,\ g))) \\
&\qquad F =\ !\mathbf{1}\ +\ (!(Vertex, Vertex, Value) \times I)
\end{aligned}
$$

この関数は，hylomorphism の性質により，catamorphism と anamorphism の合成関数に分解することができ，一般的探索関数は，

$$[\![e \triangledown acc,\ id,\ \psi]\!] = (\![e \triangledown acc]\!) \circ [\!(\psi)\!]$$

のように catamorphism $(\![e \triangledown acc]\!)$ と anamorphism $[\!(\psi)\!]$ の合成関数で表すことができる．これは，$[\!(\psi)\!]$ によって探索木を表すリストが生成され，そのリストが $(\![e \triangledown acc]\!)$ によって消費されるということを表している．hylomorphism $[\![e \triangledown acc, id, \psi]\!]$ はこの 2 つの関数が融合されたものを表している．

4.3 融合変換

この節では変換の例として，よく行われる融合変換を取り上げる．ある関数が 2 つの関数の合成関数として表されている場合には，その 2 つの関数の間では中間データの生成，受け渡しが行われるが，その 2 つの関数を 1 つの関数に融合することによって中間データの生成，受け渡しが行われないようにする変換を融合変換という．前節で定義した一般的探索関数は，hylomorphism の形で表すことができたので，hylo fusion と呼ばれる融合変換を行うことができる．hylo fusion の融合規則は，左から hylomorphism に融合する左融合規則と，右から hylomorphism に融合する右融合規則がある [7]．

左融合規則

$$f \circ \phi = \phi' \circ G\ f$$

ならば

$$f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}$$

右融合規則

$$\psi \circ g = F\ g \circ \psi'$$

ならば

$$[\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}$$

以下に，左融合規則と右融合規則の適用例を示す．

4.3.1 例 1—eccentricity

融合変換の例として，ある頂点 v から他の頂点までの距離のうちで最も大きなものを求めるという問題を考える．この例は，深さ優先探索では記述することができないので，以下で行う変換は，これまでの研究では扱えなかったものである．この問題を解くには，まず3.3.1で定義した関数*dijkstra*を用いて頂点 v から他の頂点への最短距離を求め，そこから距離のみをとりだし，そのなかの最大値を結果として返せばよい．これを行う関数*eccentricity*は以下のように定義できる．

$$
\begin{aligned}
&eccentricity : Vertex \to Graph \to Int \\
&eccentricity\ v\ g = (maximum \circ (map\ snd)) \\
&\qquad\qquad (dijkstra\ v\ g)
\end{aligned}
$$

ここで，*maximum* は引数にとったリスト中の要素の最大値を返す関数である．しかし，このプログラムは，関数合成の部分で中間データの生成，受け渡しが行われるため，効率がよくない．この中間データの生成，受け渡しを除去するために融合変換を行う．まず，3.3.1において関数*explore*を用いて記述されていた関数*dijkstra*を hylomorphism を用いて記述し直すと

$$
\begin{aligned}
&dijkstra : Vertex \to Graph \\
&\qquad\qquad \to [(Vertex, Value)] \\
&dijkstra\ v\ g = [\![e \triangledown acc, id, \psi]\!]\ (fr, g) \\
&\quad \text{where} \\
&\qquad e = [\,] \\
&\qquad acc\ ((pv, v, d), r) = (v, d)\ :\ r \\
&\qquad \psi\ (fr, Empty) = (1, ()) \\
&\qquad \psi\ ((pv, v, d) \triangleleft fr,\ (p, v, s)\ \underline{\&}\ g) \\
&\qquad\quad = (2,\ ((pv, v, d), (merge\ eval\ v\ d\ s\ fr,\ g)))
\end{aligned}
$$

$$eval\ v\ d\ x = d + weight\ (v, x)$$
$$fr = toFrontier\ [(\_, v, 0)]$$

となるので，これを用いると $eccentricity$ は，

$$eccentricity\ v\ g = (maximum \circ (map\ snd) \circ [\![e \triangledown acc, id, \psi]\!])\ (fr, g)$$

となる．次に，融合規則を適用する．左融合規則より，任意の $x : F\ [(Vertex, Value)]$ について

$$(maximum \circ (map\ snd) \circ (e \triangledown acc))\ x = ((f_1 \triangledown f_2) \circ F\ (maximum \circ (map\ snd)))\ x$$

を満たす f_1, f_2 が存在すれば，融合変換が行える．ここで，F は，

$$F = \ !\mathbf{1} + (!(Vertex, Vertex, Value) \times I)$$

である．以下で x について場合分けをすることにより，f_1, f_2 を導出する．

- $x = (1, ())$ の場合

$$\begin{array}{rcl} \text{LHS} & = & (maximum \circ (map\ snd))\ [\,] \\ & = & maximum\ [\,] \\ & = & -\infty \\ \text{RHS} & = & (f_1 \triangledown f_2) \\ & & \quad ((id + (id \times (maximum \circ \\ & & \quad\quad (map\ snd))))\ (1, ()) \\ & = & (f_1 \triangledown f_2)\ (1, ())) \\ & = & f_1\ () \end{array}$$

  よって，$f_1 = \lambda().-\infty$

- $x = (2, ((pv, v, d), xs))$ の場合

$$\begin{array}{rcl} \text{LHS} & = & (maximum \circ (map\ snd)) \\ & & \quad (acc\ ((pv, v, d), xs)) \\ & = & (maximum \circ (map\ snd)) \\ & & \quad ((v, d) : xs) \\ & = & maximum\ (d : map\ snd\ xs) \\ & = & max\ d\ (maximum\ (map\ snd\ xs)) \\ \text{RHS} & = & f_2\ ((pv, v, d), \\ & & \quad maximum\ (map\ snd\ xs)) \end{array}$$

  よって，$f_2\ ((\_, \_, d), r) = max\ d\ r$ となる．ここで，max は，引数にとった 2 つの数の最大値を返す関数である．

以上より，

$$eccentricity\ v\ g = [\![f_1 \triangledown f_2, id, \psi]\!]\ (fr, g)$$
$$\begin{array}{l} \text{where} \\ \quad f_1 = \lambda().-\infty \\ \quad f_2\ ((\_, \_, d), r) = max\ d\ r \\ \quad \psi\ (fr, Empty) = (1, ()) \\ \quad \psi\ ((pv, v, d) \triangleleft fr,\ (p, v, s)\ \underline{\&}\ g) \\ \quad\quad = (2, ((pv, v, d), (merge\ eval\ v\ d\ s\ fr, g))) \\ \quad eval\ v\ d\ x = d + weight\ (v, x) \\ \quad fr = toFrontier\ [(\_, v, 0)] \end{array}$$

となり，関数 $maximum \circ (map\ snd)$ と関数 $[\![e \triangledown acc, id, \psi]\!]$ の合成が融合された単一の関数 $[\![f_1 \triangledown f_2, id, \psi]\!]$ による表現が得られる．融合前の関数 $maximum \circ (map\ snd) \circ [\![e \triangledown acc, id, \psi]\!]$ においては，関数合成の部分で，頂点，最短距離の対のリスト，最短距離のリストが中間データとして生成され，受け渡されるが，融合後の関数 $[\![f_1 \triangledown f_2, id, \psi]\!]$ においては，これらの中間データの生成，受け渡しは行われない．

4.3.2 例 2— 強連結成分分解

この節で述べる強連結成分分解の変換例は [5] においてすでに述べられているが，適用範囲の狭い特別な変換規則を提示してそれを用いて変換を行っている．これに対して我々は，hylomorphism を経由しての変換を行うので，特別な変換規則を必要とせず，一般性の高い変換を行うことができる．グラフを強連結成分 (strongly connected component) に分解する関数 scc は，

$$\begin{array}{rcl} scc & : & Graph \to Tree\ Vertex \\ scc\ g & = & dfs\ (reverse\ (postOrd\ g)) \\ & & \quad (transposeG\ g) \end{array}$$

と定義できる [8]．関数 scc はグラフを引数にとり，そのグラフの枝の向きを関数 $transposeG$ によってすべて逆にし，そのグラフのある深さ優先探索木を，3.3.2で定義した関数 dfs を用いて求める．そのとき，関数 dfs の第 1 引数には，scc が引数にとったグラフの深さ優先探索木を後順 (post order) で渡った結果をリストにしたものが与えられる．$scc\ g$ の結果としては，仮想的な頂点 $\_$ を根とする深さ優先探索木が返ってくるが，頂点 $\_$ のそれぞれの子の中に含まれている頂点の集合が，強連結成分になっている．関数$transposeG$は

$$transposeG : Graph \to Graph$$
$$transposeG\ Empty = Empty$$
$$transposeG\ ((p, v, s)\ \&\ g) = (s, v, p)\ \&\ transposeG\ g$$

と定義され，グラフを引数にとり，そのグラフの枝の向きをすべて逆にしたグラフを結果として返す．*postOrd* は

$$postOrd : Graph \to [Vertex]$$
$$postOrd\ g = init\ (postorder\ (dfs\ (nodes\ g)\ g))$$

と定義される関数であり，グラフを引数にとり，そのグラフの深さ優先探索木を後順 (post order) で渡った結果をリストにして返す．*postorder* は,

$$postorder : Tree\ a \to [a]$$
$$postorder\ (Node\ n\ ts) = concat\ (map\ postorder\ ts)\ +\!\!+\ [n]$$

と定義される関数であり，引数に与えられた木を後順で渡った結果をリストにして返す関数である．*init* はリストを引数にとり，そのリストの最後の要素を除いたリストを返す関数であり，これにより仮想的な頂点 _ が取り除かれる．*nodes* は引数に与えられたグラフ中のすべての頂点をリストにして返す関数である．前節において*explore*を用いて記述されていた関数*dfs*を hylomorphism を用いて記述し直すと

$$dfs\ vs\ g = [\![e \triangledown acc, id, \psi]\!]\ (fr, g)\ (Node\ \_\ [\,])$$

where

$$e = id$$
$$acc\ ((pv, v, d), r) = r\ \circ\ addv\ (pv, v)$$
$$\psi\ (fr, Empty) = (1, ())$$
$$\psi\ ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g) = (2,\ ((pv, v, d), (merge\ eval\ v\ d\ s\ fr,\ g)))$$
$$eval\ v\ d\ x = d - 1$$
$$fr = toFrontier\ [(\_, v, i) | (v, i) \leftarrow (zip\ vs\ [1..\#vs])]$$

となるので，これを用いると関数*scc*は

$$scc\ g = ([\![e \triangledown acc, id, \psi]\!]\ \circ\ (id \times transposeG))\ (fr, g)\ (Node\ \_\ [\,])$$

where

$$fr = toFrontier\ [(\_, v, i) | (v, i) \leftarrow (zip\ vs\ [1..\#vs])]$$
$$vs = reverse\ (postOrd\ g)$$

となる．ここで，$\times$ は，

$$(f \times g)\ (x,\ y) = (f\ x,\ f\ y)$$

と定義される演算子を表す．右融合規則より，任意の $x : (Frontier, Graph)$ について

$$(\psi\ \circ\ (id \times transposeG))\ x = (F\ (id \times transposeG)\ \circ\ \psi')\ x$$

を満たす ψ' が存在すれば，融合変換が行える．ここで，F は,

$$F =\ !\mathbf{1}\ +\ (!(Vertex, Vertex, Value) \times I)$$

である．以下で x について場合分けをすることにより，ψ' を導出する．

- $x = (fr, Empty)$ の場合

$$\begin{aligned} \text{LHS} &= \psi\ (fr, Empty) \\ &= (1, ()) \\ \text{RHS} &= F\ (id \times transposeG) \\ &\quad (\psi'\ (fr, Empty)) \end{aligned}$$

よって，$\psi'\ (fr, Empty) = (\mathbf{1}, ())$

- $x = ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g)$ の場合

$$\begin{aligned} \text{LHS} &= \psi\ ((pv, v, d) \lhd fr, \\ &\qquad (s, v, p)\ \underline{\&}\ (transposeG\ g)) \\ &= (2, ((pv, v, d), \\ &\qquad (merge\ eval\ v\ d\ p\ fr, \\ &\qquad transposeG\ g))) \\ \text{RHS} &= F\ (id \times transposeG) \\ &\quad (\psi'\ ((pv, v, d) \lhd fr, \\ &\qquad (p, v, s)\ \underline{\&}\ g)) \end{aligned}$$

よって，

$$\psi'\ ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g) = (2, ((pv, v, d),\ (merge\ eval\ v\ d\ p\ fr,\ g)))$$

以上より，

$$scc\ g = [\![e \triangledown acc, id, \psi']\!]\ (fr, g)\ (Node\ \_\ [\,])$$

where

$$e = id$$
$$acc\ ((pv, v, d), r) = r\ \circ\ addv\ (pv, v)$$
$$\psi'\ (fr, Empty) = (1, ())$$
$$\psi'\ ((pv, v, d) \lhd fr,\ (p, v, s)\ \underline{\&}\ g) = (2,\ ((pv, v, d), (merge\ eval\ v\ d\ p\ fr, g)))$$

$$eval\ v\ d\ x = d - 1$$
$$fr = toFrontier$$
$$[(\_, v, i) | (v, i) \leftarrow (zip\ vs\ [1..\#vs])]$$
$$vs = reverse\ (postOrd\ g)$$

となり, 関数 $[\![e \triangledown acc, id, \psi]\!]$ と $(id \times transposeG)$ の合成が融合された単一の関数 $[\![e \triangledown acc, id, \psi']\!]$ による表現が得られる. 融合前の関数 $[\![e \triangledown acc, id, \psi]\!] \circ (id \times transposeG)$ においては, 関数合成の部分で枝の向きがすべて逆向きにされたグラフが中間データとして生成され, 受け渡されるが, 融合後の関数 $[\![e \triangledown acc, id, \psi']\!]$ においては, その中間データの生成, 受け渡しは行われない.

5 一般的探索関数の実現

この節では, 3.2節で定義した一般的探索関数 *explore* の関数型言語 Haskell による効率のよい 1 つの実現法を示し, 計算量, 融合変換の効果について調べる.

5.1 一般的探索関数の計算量

ここでは, 一般的探索関数 *explore* の実現法を示す前に, 関数 *explore*

$$explore\ acc\ e\ eval\ (fr, g)$$

の計算量をいくつかのパラメータを用いて記述しておく. グラフ g の頂点数を N, 枝数を M, acc の平均計算量を C_{acc}, 前線への $\uplus$ による要素の挿入の平均計算量を $C_{\uplus}$, 前線のパターンマッチ $(pv, v, d) \triangleleft fr$ にかかる平均計算量を $C_{\triangleleft}$, グラフのアクティブパターンマッチングにかかる平均計算量を $C_{\underline{\&}}$ とする. 関数 *eval* の計算量は入力グラフのサイズに依存しないので $O(1)$ である. 前線への要素の挿入は $O(M)$ 回, 他の演算は $O(N)$ 回行われるので, 関数 *explore* は

$$O(MC_{\uplus} + N(C_{\triangleleft} + C_{acc} + C_{\underline{\&}}))$$

のコストで実現可能である.

5.2 前線の実現

前線は, 評価値を優先度とする優先順位付き待ち行列 (priority queue) であるので, 一般の場合にはヒープ木を用いて実現すればよい. 深さ優先探索, 幅優先探索の場合には, 評価値の求め方があらかじめ決まっているので, 前線の実現をそれぞれに特化したものにすることができる. 具体的には, 深さ優先探索の場合にはスタック, 幅優先探索の場合には両方向出し入れ可能なキュー, の関数型言語での実現法を応用することにより, それぞれの性質に応じた実現をすることができる. 以下で, 一般の場合, 深さ優先探索の場合, 幅優先探索の場合について, 前線と前線操作関数の実現法, 前線操作関数の計算量について述べる. 前線操作関数は以下の 3 つの関数とする.

- 前線を引数にとり, 前線が空のとき `True`, 空でないとき `False` を返す関数
  ```
  isEmpty :: Frontier -> Bool
  ```
- 要素 1 つを前線に加える関数
  ```
  insert :: (Vertex,Vertex,Value)
                  -> Frontier -> Frontier
  ```
- 空でない前線を引数にとり, 最小要素と残りの前線との対を返す関数
  ```
  deletemin :: Frontier ->
        ((Vertex,Vertex,Value),Frontier)
  ```

ただし, 関数 `deletemin` は評価値最小の要素の取りだしのみを行い, その評価値最小要素の第 2 要素と同じ頂点を第 2 要素に持つ要素すべてを取り除くということは行わないものとする. この `deletemin` の定義では, 訪問済みの頂点を第 2 要素に持つ要素が前線中に存在しうるようになるので, 評価値最小要素を取り出す際には, その第 2 要素が訪問済みの頂点であるかどうかを調べる必要があるが, これは別の関数で調べるものとする.

関数 `insert` が $\uplus$ に対応し, この平均計算量が $C_{\uplus}$ である. 関数 `deletemin` の平均計算量を C_{dm} とすると, 前線への要素の挿入が $O(M)$ 回行われることより関数 `deletemin` は $O(M)$ 回呼び出されることになるので,

$$C_{\triangleleft} = O(MC_{dm}/N)$$

である. 関数 `isEmpty` の計算量は, 以下のいずれの場合においても $O(1)$ で実現し, 呼び出されるのは $O(M)$ 回なので, 無視することができる.

- 一般の場合

  一般の場合には前線はヒープ木を用いて実現すればよい. ヒープ木の型は `Htree a` とし, `Htree a` は型 `a` の要素を持つヒープ木の型であるとする. ヒープ木の操作関数として以下の 3 つを定義する.

  - ヒープ木が空であるとき `True`, 空でないとき `False` を返す関数

```
isEmptyHtree :: Htree a -> Bool
```

– ヒープ木に要素を1つ加える関数

```
insertHtree :: a -> Htree a
                 -> Htree a
```

– 空でないヒープ木を引数にとり，最小要素と残りのヒープ木との対を返す関数

```
deleteminHtree :: Htree a
                    -> (a,Htree a)
```

以上の3つの関数は，

– 関数 isEmptyHtree は $O(1)$ で，
– 関数 insertHtree は $O(\log(M))$ で，
– 関数 deleteminHtree は $O(\log(M))$ で，

それぞれ実現可能である [1]．このヒープ木およびヒープ木の操作関数を用いることにより，前線は以下のように定義できる．

```
type Frontier =
    Htree (Vertex,Vertex,Value)
isEmpty = isEmptyHtree
insert = insertHtree
deletemin = deleteminHtree
```

前線操作関数の計算量は，isEmpty は $O(1)$，あとの2つの関数は

$$C_{\uplus} = O(\log(M)),$$
$$C_{\mathsf{dm}} = O(\log(M))$$

となるので，関数 *explore* は，

$$O(M \log(M) + N(C_{acc} + C_{\underline{\&}}))$$

のコストで実現可能である．入力グラフに並列枝がない場合には，枝数 $M = O(N^2)$ であるので，関数 *explore* は

$$O(M \log(N) + N(C_{acc} + C_{\underline{\&}}))$$

のコストで実現可能である．

- 深さ優先探索の場合

スタックを応用して前線を実現することにより，深さ優先探索の場合に特化した実現にすることができる．関数型言語では，リストを1つ用いてスタックを実現する．スタックの型は

```
type Stack a = [a]
```

とし，[a] は型 a の要素を持つリストの型を表す．空のスタックは空リスト [] とする．スタックの操作関数として以下の4つを定義する．

– スタックが空であるとき True，空でないとき False を返す関数

```
isEmptyStack :: Stack a -> Bool
```

– スタックに要素を1つ加える関数

```
insertStack :: a -> Stack a
                 -> Stack a
```

– 空でないスタックを引数にとり，先頭要素を返す関数

```
headStack :: Stack a -> a
```

– 空でないスタックを引数にとり，先頭要素を除いた残りのスタックを返す関数

```
tailStack :: Stack a -> Stack a
```

以上の4つの関数は，

– 関数 isEmptyStack は $O(1)$ で，
– 関数 insertStack は $O(1)$ で，
– 関数 headStack は $O(1)$ で，
– 関数 tailStack は $O(1)$ で，

それぞれ実現可能である．このスタックおよびスタックの操作関数を用いて前線を実現する．まず，前線の型は

```
type Frontier =
    Stack (Vertex,Vertex,Value)
```

とする．この，前線を実現しているスタックが，そのスタック中の要素が常に評価値の小さい順にならんでいるという性質を持つように，前線操作関数を定義する．

```
isEmpty = isEmptyStack

insert x@(pv,v,d) fr =
  if isEmpty fr then
    insertStack x fr
  else
    let x'@(pv',v',d') =
                headStack fr
    in if d<=d' then
         insertStack x fr
       else
         insertStack x'
           (insert x (tailStack fr))
```

```
deletemin fr = (headStack fr,
                tailStack fr)
```

前線操作関数の計算量は，**isEmpty**，**deletemin** は $O(1)$ となる．**insert** は，深さ優先探索関数の場合には，要素は常にリストの先頭に挿入されるので，1回の **insert** の計算量は，$O(1)$ である．よって，

$$C_{\uplus} = O(1),$$
$$C_{\mathrm{dm}} = O(1)$$

となるので，深さ優先探索の場合，関数 *explore* は，

$$O(M + N(C_{acc} + C_{\underline{\&}}))$$

のコストで実現可能である．

- 幅優先探索の場合

両方向出し入れ可能なキューを応用して前線を実現することにより，幅優先探索の場合に特化した実現にすることができる．関数型言語では，リストを2つ用いて両方向出し入れ可能なキューを実現する [11]．両方向出し入れ可能なキューの型を

```
type DEQueue a = ([a],[a])
```

とする．空のキューは `([],[])` とする．キューの操作関数として以下の6つを定義する．先頭からの要素挿入関数は前線操作関数で必要とされないので要素挿入関数は末尾からのもののみ定義する．

- キューを引数にとり，キューが空のとき **True**，キューが空でないとき **False** を返す関数

```
isEmptyDEQueue :: DEQueue a
                   -> Bool
```

- キューに要素を末尾から1つ加える関数

```
snoc :: a -> DEQueue a
          -> DEQueue a
```

- 空でないキューを引数にとり，キューの先頭要素を返す関数

```
headDEQueue :: DEQueue a -> a
```

- 空でないキューを引数にとり，その先頭要素を除いたキューを返す関数

```
tailDEQueue :: DEQueue a
                -> DEQueue a
```

- 空でないキューを引数にとり，キューの末尾要素を返す関数

```
lastDEQueue :: DEQueue a -> a
```

- 空でないキューを引数にとり，その末尾要素を除いたキューを返す関数

```
initDEQueue :: DEQueue a
                -> DEQueue a
```

関数 **isEmptyDEQueue**，**headDEQueue**，**lastDEQueue** は $O(1)$ で，**snoc**，**initDEQueue**，**tailDEQueue** は，平均計算量 $O(1)$ で実現可能である [11]．このキューおよびキューの操作関数を用いて前線を実現する．まず，前線の型は，

```
type Frontier =
     DEQueue (Vertex,Vertex,Value)
```

とする．この，前線を実現しているキューが，そのキュー中の要素が常に評価値の小さい順にならんでいるという性質を持つように，前線操作関数を定義する．

```
isEmpty = isEmptyDEQueue

insert x@(pv,v,d) fr =
  if isEmpty fr then
    snoc x fr
  else
    let x'@(pv',v',d') =
             lastDEQueue fr
    in if d>=d' then
         snoc x fr
       else
         snoc x'
         (insert x (initDEQueue fr))

deletemin fr =
   (headDEQueue fr, tailDEQueue fr)
```

前線操作関数の計算量は，**isEmpty**，**deletemin** は $O(1)$ となる．幅優先探索の場合には，要素は常にキューの末尾に挿入されるので，**insert** の平均計算量は $O(1)$ となる．よって，

$$C_{\uplus} = O(1),$$
$$C_{\mathrm{dm}} = O(1)$$

となるので，幅優先探索の場合，関数 *explore* は

$$O(M + N(C_{acc} + C_{\underline{\&}}))$$

のコストで実現可能である．

5.3 アクティブパターンマッチングの実現

ここでは，グラフのアクティブパターンマッチングを $C_{\underline{\&}} = O(M/N)$ で実現する方法を示す．この方法を用いることにより，関数 *explore* は

$$O(MC_{\uplus} + N(C_{\triangleleft} + C_{acc}))$$

のコストで実現可能となる．以下で述べる方法は，[5]で行われている，深さ優先探索関数の ML による実現法にも用いられている．

まず，グラフは書き換えのできない，隣接リストの配列として表現する．

```
type Graph = Array Vertex
                          [(Vertex, Weight)]
```

この宣言の右辺は，Vertex 型の要素を添字とし，[(Vertex,Weight)] 型の要素を値とする，書き換えのできない配列型を表している．配列の値は，各頂点を始点とする枝の先の頂点 (successor) とその枝の重みの対のリストを表すものとする．アクティブパターンマッチングで要求されるのは，ある頂点の，その時点での successor のリストと，その頂点とそれに付随する枝を除いた結果，残ったグラフである．そのために，頂点を訪問するごとに訪問済みの印をつけるようにし，各時点におけるグラフを，探索対象グラフ全体を表している書き換えのできない配列と，訪問済み情報との対で表現する．そして，アクティブパターンマッチングの際には，探索対象グラフ全体を表している配列と，訪問済みの情報と，現在訪問中の頂点とを用いてその時点での successor のリストを得，また，現在訪問中の頂点に訪問済みの印をつけることによって残りのグラフを得る．これを効率よく行うために，各頂点を添字とする書き換え可能な配列を 1 つ用いて，訪問済みの情報を保持することとし，state transformer [10] [12]の技法により，この書き換え可能な配列を状態として保持し，それを次々と受け渡していくようにする．このような方法は，[8]の深さ優先探索関数の実現においても用いられている．以上で述べた方法でアクティブパターンマッチングを行う関数を実現すると，

```
apm :: Vertex -> (Set s, Graph)
        -> ST s [(Vertex,Weight)]
apm v (m,g) = do {
                s <- suc v (m,g);
                include m v;
                return s
              }
```

となる．do { ... } は do expression であり，state transformer 等のモナドを読みやすく記述するためのものである．関数 apm の第 1 引数には訪問中の頂点，第 2 引数には，訪問済みの情報が保持されている書き換え可能な配列と探索対象グラフ全体を表している配列との対が与えられる．apm v (m,g) においては 2 つのことが行われる．まず，s <- suc v (m,g) によって，頂点 v の，グラフ (m,g) における successor のリストが得られ，それに s がバインドされる．次に，include m v によって，配列 m の，添字 v の値が訪問済みを表す値になる．include m v は，配列 m 中の 1 つの要素の値を書き換えるということを行うのでこれの 1 回の計算量は $O(1)$ である．suc v (m,g) においては，まず配列 g の添字 v の値を求める．それはグラフ g 中の頂点 v の successor を表しており，その中の各頂点を添字として配列 m の値を参照し，訪問済みの頂点を除くことによって，グラフ (m,g) における，頂点 v の successor を得る．このような successor の求め方は，[5]の深さ優先探索関数の ML による実現においても行われている．この関数 suc は，次のように定義することができる．

```
suc :: Vertex -> (Set s, Graph)
       -> ST s [(Vertex,Weight)]
suc v (m,g) = suc' m (g!v)

suc' :: Set s -> [(Vertex,Weight)]
        -> ST s [(Vertex,Weight)]
suc' m (x:xs) =
  do {
    visited <- contains m (fst x);
    if visited then
      suc' m xs
    else do {
      ys <- suc' m xs;
      return (x : ys)
    }
  }
suc' m [] = return []
```

suc v (m,g) の 1 回の計算量は，グラフ g の添字 v に

ける値のリスト g!v の長さを c とすると, $O(c)$ である. 関数 apm の第1引数には毎回異なる頂点が与えられ, すべての頂点について呼び出されるので, suc v (m,g) の合計の計算量は, $O(M)$ となる. また, アクティブパターンマッチングを行う関数 apm は $O(N)$ 回呼ばれるので, 関数 include は $O(N)$ 回呼ばれる. include m v の1回の計算量が $O(1)$ であることより, include m v の合計の計算量は $O(N)$ となる. よって, apm v (m,g) の合計の計算量は $O(M)$ であり, apm v (m,g) は $O(N)$ 回呼び出されるので, 平均をとると, $C_{\underline{\&}} = O(M/N)$ を達成していることが分かる.

5.4 一般的探索関数の実現

5.3節で定義した, グラフのアクティブパターンマッチングを $C_{\underline{\&}} = O(M/N)$ で行う関数 apm を用いて一般的探索関数 *explore* を関数型言語 Haskell により実現すると, 図3のようになる. 関数explore は, まず, 頂点の訪問済み情報を保持するための, グラフ g の頂点数の長さの配列を mkEmpty (bounds g) によって生成し, m をそれにバインドし, それと, グラフ全体を表す書き換えのできない配列 g との対 (m,g) をグラフとして, 関数 exploreaux を呼び出す. runST は state transformer を引数にとり, それを初期状態に適用させ, その結果から最終状態を捨てたものを結果として返す関数を表す. mkEmpty は書き換え可能な配列を作成する関数を表し, 引数に配列の添字の範囲をとり, その大きさの配列を作成し, それを state transformer にしたものを結果として返す. この計算量は, 作成する配列の長さを l とすると, $O(l)$ である. bounds は書き換えのできない配列を引数にとり, その配列の添字の範囲を返す. この計算量は, 引数に与えられる配列の長さを l とすると, $O(l)$ である. 関数 exploreaux では, まず, isEmpty fr によって, 前線 fr が空かどうかを調べ, 空ならば e を state transformer にしたものを結果として返し, 空でないならば deletemin fr によって前線 fr から, 評価値最小の要素 (pv,v,d) を取り出し, 残った前線を fr' とする. 次に, 頂点 v が訪問済みかどうかを contains m v によって調べ, 訪問済みならば visited は True, そうでないならば visited は False となる. 頂点 v が訪問済みだった場合は, fr' を前線として, exploreaux を呼び出す. 頂点 v が訪問済みでなかった場合は, s <- apm v (m,g) によって, 頂点 v の, グラフ (m,g) における successor のリストが得られ, それに s がバインドされ, そして, 頂点 v が訪問済みになる. 次に, exploreaux を呼びだすことによって再帰部分の値を得, それに x がバインドされる. exploreaux を呼びだすときに, 前線としては, s <- apm v (m,g) によって得られた successor のリスト s を関数 merge によって前線 fr' に加えたものを与える. 最後に, acc ((pv,v,d),x) によって非再帰部分 (pv,v,d) と再帰部分の値 x を用いてある計算が行われ, それを state transformer にしたものが結果として返される. return は, 引数にとった値を state transformer にしたものを結果として返す関数を表す.

5.5 実験

ここでは, 最短路問題を解くダイクストラ法を例として, 実際に効率のよい実現になっているかどうかを確認する. また, 4.3.1節で2つの例について融合変換を行ったが, ここでは, 関数 *eccentricity* の融合変換の効果を確認する.

関数 *dijkstra* の計算量は, $C_{acc} = O(1)$ であるので, $O(M\log(M)+N)$ となる. グラフに並列枝がない場合には計算量は $O(M\log(N)+N)$ となる. これは, 手続き型言語においてヒープを用いてダイクストラ法を実現したときの計算量と次数が同じである. 実際にこの計算量になっているかどうかを以下で調べる. 5.4節で記述した関数explore を用いて最短距離を求める関数dijkstra を記述し, ある頂点から他のすべての頂点への最短距離をいくつかのグラフについて求めるということを, 関数型言語 Haskell のインタプリタ hugs1.4 上で行った. 入力として与えるグラフは, 並列枝のない連結グラフとし, ランダムに生成した. いくつかのグラフに対して, 簡約段数を調べ, 頂点数 N, 枝数 M, 簡約段数 R, 簡約段数 R と $M\log(N)$ の比 $R/(M\log(N))$ を表にしたものを表1に示す. これをみると, $R/(M\log(N))$ がほぼ一定であり, 簡約段数 R がほぼ $M\log(N)$ に比例していることを読みとることができる.

以下で関数 *eccentricity* の融合変換の効果について述べる. 5.4節で記述した関数explore を用いてある頂点から他の頂点までの距離のうちで最も大きなものを

```
explore :: (((Vertex,Vertex,Value),a) -> a) -> a
           -> (Vertex -> Value -> (Vertex,Weight) -> Value)
           -> (Frontier, Graph) -> a
explore acc e eval (fr,g) =
    runST (do {m <- mkEmpty (bounds g);
               exploreaux acc e eval fr (m,g)})

exploreaux :: (((Vertex,Vertex,Value),a) -> a) -> a
              -> (Vertex -> Value -> (Vertex,Weight) -> Value)
              -> Frontier -> (Set s, Graph) -> ST s a
exploreaux acc e eval fr (m,g) =
    if isEmpty fr then
      return e
    else
      let ((pv,v,d),fr') = deletemin fr
      in do {
        visited <- contains m v;
        if visited then
          exploreaux acc e eval fr' (m,g)
        else do {
          s <- apm v (m,g);
          x <- exploreaux acc e eval (merge eval v d s fr') (m,g);
          return (acc ((pv,v,d),x))
        }
      }

merge :: (Vertex -> Value -> (Vertex,Weight) -> Value) -> Vertex
         -> Value -> [(Vertex,Weight)] -> Frontier -> Frontier
merge eval v d [] fr = fr
merge eval v d (x:s) fr = let d' = eval v d x
                          in merge eval v d s (insert (v,(fst x),d') fr)
```

図 3 一般的探索関数 explore

求める関数**eccentricity**を記述し, 融合変換前と融合変換後について, 関数型言語 Haskell のインタプリタ hugs1.4 上で, 簡約段数, セル使用数を調べた. 引数には, **dijkstra** の実験で与えたグラフと同じものを与えた. 頂点数を N, 枝数を M, 簡約段数を R, セル使用数を C としてこれを表にしたものが表 2である. これを見ると, 融合変換によって, 簡約段数, セル使用数ともに減少していることを読みとることができる.

最後に, 本論文で取り上げた例のうちのいくつかの関数の計算量について述べておく. 4.3.1節で記述した関数 *eccentricity* の計算量は,

$$maximum \circ (map\ snd)$$

の計算量が $O(N)$ であるので, 全体の計算量は *dijkstra* と同じで, $O(M\log(M)+N)$ となる. 次に, 4.3.2節

表 1 `dijkstra` の簡約段数

頂点数 N	枝数 M	簡約段数 R	$R/(M\log(N))$
10	38	3630	41
50	237	33220	36
100	934	154793	36
200	1933	369657	36
400	3935	875620	37
600	5953	1472822	39
800	7936	2075312	39
1000	9939	2756324	40

表 2 `eccentricity` の融合変換の効果

頂点数 N	枝数 M	融合変換前		融合変換後	
		簡約段数 R	セル使用数 C	簡約段数 R	セル使用数 C
10	38	3548	7793	3465	7551
50	237	32818	73328	31395	69106
100	934	153991	344191	148643	328244
200	1933	368055	829807	347357	767910
400	3935	872418	1993813	791020	1750016
600	5953	1468020	3387919	1285922	2842222
800	7936	2068910	4830634	1746112	3863037
1000	9939	2748322	6465660	2244824	4956163

で記述した関数 scc の計算量を求めておく．まず，関数 dfs の計算量は，一般の場合の前線の実現法を用いると $O(M\log(M)+NC_{acc})$ であるが，深さ優先探索に特化した前線の実現法を用いると，$O(M+NC_{acc})$ となる．関数 $addv$ の計算量が $O(M)$ であるので $C_{acc}=O(M)$ であるが，木を配列を用いて実現することにより関数 $addv$ を $O(1)$ で実現することができ，$C_{acc}=O(1)$ とすることができる．よって，関数 dfs は $O(M+N)$ で実現可能である．また，関数 $transposeG$ の計算量は $O(M+N)$，関数 $postOrd$ の計算量は $O(M+N)$，関数 $reverse$ の計算量は $O(N)$ であるので，関数 scc は $O(M+N)$ で実現可能である．

6 結論

本論文では，グラフアルゴリズムを自然に記述し，そしてプログラム変換によってその効率を改善する手法を提案した．本論文の主な貢献は以下のようにまとめられる．

- グラフの一般的探索を行う関数の再帰的定義を示した．この一般的探索関数を用いることにより，深さ優先探索，幅優先探索を用いるアルゴリズムを含むさまざまなグラフアルゴリズムを自然に記述することができる．
- これまでに行われていたグラフ上の関数のプログラム変換では，独自に定義した深さ優先探索関数等についてのみの変換であったが，本論文では一般的探索関数を hylomorphism と呼ばれる形に変換し，その上で変換を行った．これにより，すでによく知られている枠組で，一般性のある各種の変換を行うことができ，本稿では例として融合変換を行った．
- 一般的探索関数は効率のよい実現をすることが可能であり，state transformer を用いる 1 つの効率のよい実現法を示した．

また，今後の課題としては，以下のようなことが考えられる．

- 本論文で定義した一般的探索関数の hylomorphism 表現において，それを，anamorphism と catamorphism の合成関数に分離したときに，その間で受け渡される中間データは，探索木を表すリストであったが，この中間データの構造がリスト構造以外になるようにすることも可能だと思われる．
- 本論文では，探索によって表されるアルゴリズムを一般的に取り扱った．この範囲外に属していると思われる，最大マッチング問題，ハミルトン閉路問題，平面性判定問題等の関数型言語での取り扱いについて，今後調べていく予定である．

参考文献

[1] Bird, R. : *Introduction to Functional Programming using Haskell (second edition)*, Prentice Hall, 1998.

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

[3] Dijkstra, E. W. : A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik*, Vol. 1 (1959), pp. 269–271.

[4] Erwig, M. : Graph Algorithms = Iteration + Data Structures?, *the 18th International Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, 1992, pp. 277–292.

[5] Erwig, M. : Functional Programming with Graphs, *2nd ACM SIGPLAN International Conference on Functional Programming(ICFP '97)*, 1997, pp. 52–65.

[6] Gibbons, J. : An Initial-Algebra Approach to Directed Acyclic Graphs, *Mathematics of Program Construction*, LNCS 947, 1995, pp. 282–303.

[7] Hu, Z., Iwasaki, H. and Takeichi, M. : Deriving Structural Hylomorphisms from Recursive Definitions, *ACM SIGPLAN International Conference on Functional Programming(ICFP '96)*, 1996, pp. 73–82.

[8] King, D. J. and Launchbury, J. : Structuring Depth-First Search Algorithms in Haskell, *Conference record of POPL '95: the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, 1995, pp. 344–354.

[9] Launchbury, J. : Graph Algorithms with a Functional Flavour, *1st International Spring School on Advanced Functional Programming*, LNCS 925, 1995, pp. 308–331.

[10] Launchbury, J. and Peyton Jones, S. L. : Lazy Functional State Threads, *PLDI '94. Proceedings of the ACM SIGPLAN '94 conference on Programming Language Design and Implementation*, 1994, pp. 24–35.

[11] Okasaki, C. : *Fundamentals of Amortization*, Cambridge University Press, 1998.

[12] Peyton Jones, S. L. and Wadler, P. : Imperative Functional Programming, *Conference recored of the twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993, pp. 71–84.

[13] Prim, R. C. : Shortest Connection Networks and some Generalizations, *The Bell System Technical Journal*, Vol. 36 (1957), pp. 1389–1401.

[14] Ravelo, J. N. : A Class of Graph Algorithms, 1996. Qualifying dissertation submitted in application for transfer to DPhil. status, Oxford University Computing Laboratory.

[15] Sheard, T. and Fegaras, L. : A Fold for All Seasons, *Conference on Functional Programming Languages and Computer Architecture*, 1993, pp. 233–242.

[16] Takano, A. and Meijer, E. : Shortcut Deforestation in Calculational Form, *Conference on Functional Programming Languages and Computer Architecture*, 1995, pp. 306–313.

[17] 篠埜功，胡振江，武市正人：構成的手法によるグラフアルゴリズムの導出，日本ソフトウェア科学会第 *15* 回大会論文集，1998, pp. 269–272.

[18] 石畑清：岩波講座ソフトウェア科学 *3* アルゴリズムとデータ構造，岩波書店，1989.