A-012

動的な階層構造に基づくグラフ書き換え言語 REGREL+

Graph Rewriting Language REGREL+ based on Dynamic Hierarchical Structure

東 達軌[†] Tatsuki Higashi
武田 正之[‡] Masayuki Takeda

1 はじめに

書換え系では書換え規則の適用順序を決定する機構が，その系の性質を大きく決定付ける．グラフ書換え系 REGREL[1] では，分類と階層と呼ばれる制御構造に対してトップダウンに書換え規則の適用を進める．階層は分岐をもたない直列に配置された構造であり，書換え対象と書換え規則を含む．計算は，ある階層に含まれる書換え規則を，一つ下位の階層に含まれる書換え対象や書換え規則に適用することで進めていく．この構造は，上位に高階書換え規則を配置し，それに由来する特殊化が下位の階層へ伝搬していくプロセスを表現するには適している．しかし，新たな階層の挿入や，階層の分岐は認められていないため，書換え対象の計算順序制御には適していない．たとえば，あるグラフの部分グラフを取り出して，その部分にだけ特定の書換え規則群を適用するなどの適用順序制御を表現しづらい．一方，LMNtal[2, 3] の基盤となっている Membrane Computing[4] では，膜と呼ばれる制御構造に対してボトムアップに書換え規則の適用を行っていく．膜は書換え対象と書換え規則，膜自身を含む木構造を形成する．膜構造を用いた計算は，ある膜に含まれる書換え規則によって膜が含む要素を書換える．この構造は，書換え対象の要素に対する計算順序の制御方法として優れている．しかし，膜構造は木構造をなすため，全ての膜に作用する書換え規則などは表現しにくい．また，REGREL で行うような高階書換え規則の適用順序制御には向かない．

そこで本論文では，書換え対象の書換えと，高階書き換えなどメタプログラミング両方に対して利用しやすい書換え規則の適用順序制御機構として sort を提案する．sort は親子関係に基づく閉路のないグラフ構造を持つ構造である．それぞれの sort は書換え規則と書換え対象を含む．この sort 構造に基づいて祖先からトップダウンに書換え規則を適用する．この点では REGREL における分類と階層による制御と同様であるが，書換え対象に対する動的な計算順序制御も簡潔に記述することができる．また，sort 構造自体の操作も sort 構造の下で表現することが可能である．

以下に本論文の構成を示す．まず sort 構造の定義を示す．次にその sort の階層構造を計算順序制御機構とする新しいグラフ書き換え言語である REGREL+ の定義を示す．そして，REGREL+ の元での sort の階層構造の操作方法とプログラミング手法について論じる．最後に既存の計算順序制御機構との比較を行う．

2 Sort

sort は書換え規則の適用範囲の限定と，書換え規則の適用順序を決定するための機構である．

定義 2.1 sort

sort は以下のように構成される構造である．

- 固有の名前を 1 つだけ持つ
- 親と子の sort を有限個数持つ

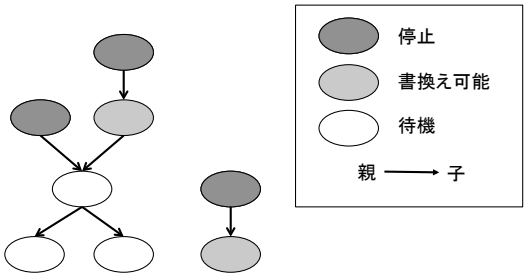

図 1 sort による階層構造

- 書換え対象と書換え規則を有限個数含む

sort はその親子関係からなるグラフ構造を形成するが，閉路が存在してはならない．sort に含まれる書換え対象は文字列やグラフなど，書換え系によって異なる．なお，ある書換え対象や書換え規則は複数の sort に含まれていても良い．

2.1 書換え規則の適用順序

適用順序は各 sort の状態を用いて決定する．

定義 2.2 sort の状態

sort は以下の 3 つの状態のうち 1 つを取る．

- 停止: 書換えが停止した sort
  以下のどちらかを満たすとき停止しているとする
  - 親を持たない
  - 親が持つ全ての書換え規則を適用する事ができない
- 書換え可能: 書換え対象となる sort
  以下の両方を満たす場合に書換え可能であるとする
  - すべての親が停止している
  - 親の持つ書換え規則を適用可能なグラフが存在する
- 待機: 停止しておらず書換え可能でもない

この状態の定義から，まず親を持たない sort が停止状態になる．そして停止した sort に含まれる書換え規則が，書換え可能な子の sort に含まれる書換え対象を書換える．書換えが適用できなくなったら書換え可能から停止状態へと遷移する．最後にすべての sort で書換えが停止したとき，系全体の書換えが停止したとみなす．ただし，書換え対象の追加や，他の sort での書換えによって停止状態 sort が書換え可能や待機状態になることがある．

この sort 構造の利点として，どの sort も有限個の親を取ることができる点がまず挙げられる．例えば，すべての sort の親となる sort を用意することで，子となる sort に対して横断的に適用される書換え規則を表現することができる．このような表現は膜，分類と階層による計算制御では表現しにくい．次に，書換え対象として書換え規則を含む高階書換え系に関しては，高階書換えが祖先から順に行われその影響が子へと波及していく動作を直接的に表現可能な点が挙げられる．構造が木構造で，共通の祖先に当たる膜を作成できない膜計算ではこの動作は表現しにくい．そして，sort の動的な追加や削除，親子関係の追加削除を認めるならば，書換え対象の書換え順序の制御に用いることが可能な点も挙げられる．

† 東京理科大学大学院 理工学研究科 情報科学専攻
Department of Information Sciences, Graduate School of Science and Technology, Tokyo University of Science
‡ 東京理科大学 理工学部 情報科学科
Department of Information Sciences, Tokyo University of Science

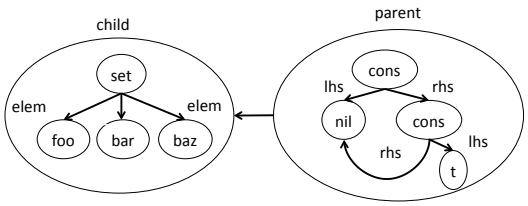

図 2　sort と REGREL グラフの例

2.2　結果の返戻

系の書換えが停止した際に，ある sort S_R に含まれる全ての書換え対象を計算結果とする．その sort S_R が何らかの理由で消滅した場合は計算失敗とする．

3　プログラミング言語 REGREL+

ここでは sort の階層構造に基づくグラフ書換え言語 REGREL+ の定義の概要を示す．REGREL+ は REGREL を基本とし，その計算順序制御構造を分類と階層から sort に変更したものである．紙面の関係上，グラフ構造や書換え規則の適用など詳細は文献 [1] に譲り，sort に関連して追加，変更のあった仕様について主に論じる．

3.1　REGREL グラフ

REGREL+ で扱われるグラフは REGREL グラフと呼ばれ，以下のように定義される．

定義 3.1 REGREL グラフ
ラベル全体の集合 L のもとで，REGREL グラフ G を以下のように定義する．

- $V(G)$: G に含まれる頂点の有限集合
  - $label_G : V(G) \to L$ 頂点の持つラベル
- $E(G) = \{(v_s, a, v_d) \mid v_s, v_d \in V(G), a \in L\}$: G に含まれる接続と呼ばれるラベル付き有向辺の有限集合．
  - v_s, a, v_d はそれぞれ接続の始点，ラベル (属性)，終点である
  - 相異なる接続 $(v_s, a, v_d), (v_s, a', v_d) \in E(G)$ に関して $a \neq a'$
- $S(G)$: G に含まれる sort の有限集合
  - $name_G : S(G) \to L$ sort の持つ名前
  - $roots_G : S(G) \to \mathfrak{P}(V(G))$ sort が参照する頂点
  - $children_G : S(G) \to \mathfrak{P}(S(G))$ sort の子の集合

接続 (v_s, a, v_d) の持つラベル a は属性と呼ばれる．以下誤解の無い範囲で，ラベル $s \in L$ を持つ頂点を単に s 頂点，属性 $a \in L$ を持つ接続を単に a 接続と呼ぶ．また誤解のない範囲で，$label_G$ を単に $label$ と略記する．

定義 3.2 根付き連結グラフ
REGREL グラフ G における根付き連結グラフ $rgraph_G(v)$ とは，根となる頂点 v から有向辺を辿って到達可能な全ての頂点（根自身を含む）と，その頂点間の全ての接続からなるグラフを指すものとする．

以下誤解の無い範囲で根付き連結グラフを単にグラフと呼ぶ．

定義 3.3 sort に所属するグラフ
ある sort $s \in S(G)$ に所属するグラフは $\{rgraph_G(v) | v \in roots_{S(G)}\}$ である．

3.2　項表現

REGREL グラフを文字列表記で表すための項表現を図 3 に示す．ただし，終端記号は引用符で囲う形式でなく，下線付きの語として表現する．

```
Structure  ::=  SortElems ;
Sort       ::=  [SortName] { SortElems }
SortName   ::=  # Name
SortElems  ::=  [SortElem { , SortElem }]
SortElem   ::=  Sort | Graph | Name | SortName
Graph      ::=  Root | (Root {, Root})
Root       ::=  (( Root ) | Rule | Node )
Node       ::=  Name [[ Name ]] [(ArcList)]
Name       ::=  var [ [ label ] ] | label
ArcList    ::=  (Arc | Comp) {, (Arc | Comp)}
Arc        ::=  Name(: | :/) Root
Comp       ::=  Arc | Arc in Node
Rule       ::=  [Root] [,PredList] -> [Root] | (Rule)
PredList   ::=  Pred {, Pred}
Pred       ::=  Root op Root
```

図 3　REGREL グラフの項表現

- Sort は sort 名 SortNmae, sort 要素 SortElems で構成され，以下の sort s を表す
  - sort 名 n_s が与えられた場合
    - 名前 n_s を持つ sort が存在するならばその sort
    - 名前 n_s を持つ sort が存在しないならば，名前 n_s を持つ新しい sort
  - 名前が指定されなかった場合は，ユニークな名前を持つ新しい sort

  sort s が含むグラフと，子となる sort は SortElems で表される
  - Graph $rgraph_G(v)$ が含まれていならば，$v \in Roots_G(s)$．ただし G は系全体のグラフである
  - Sort s_c が含まれているならば，$s_c \in Children_G(s)$
- Node は変数名，ラベル，接続 Arc の列で構成され，以下の頂点 v_s を表す
  - 頂点に変数名 var が与えられていない場合
    - ラベルを持つ: そのラベルを持つ固有の頂点
    - ラベルを持たない ([] 形式): ユニークな名前を持つ固有の頂点
  - 変数名が与えられている場合は，他の同じ変数名を持つ Node と同じ頂点を共有するものとする．

  接続の列に，属性 a, 終点 v_d を持つ Arc が現れる場合，接続 (v_s, a, v_d) が追加される
- 変数 var は頂点，接続，sort の参照を表現する
  - 変数名はある大文字英字から始まる文字列である
- ラベル label は頂点，sort が所有する要素である
  - ラベル名は小文字英字，数字，記号から始まる文字列である
  - アンダーバーから始まるラベル名は REGREL+ で予約されたラベル名であるので，基本的には使用するべきではない

なお変数のスコープは Structure で閉じている．

例 3.1 sort 構造と項表現

```
#parent{cons(lhs:N[nil], rhs:cons(lhs:t, rhs:N)),
  #child{set(elem:foo, elem:bar, elem:baz)} };
```

図 4　項表現の例

図 4 は図 2 に対応する項表現である．

parent sort に含まれるグラフは線形リストを表している．線形リストはノードを表す cons 頂点と，終端を表す nil 頂点によって表現する．cons 頂点は要素への lhs 接続，次のノードへの rhs 接続をもつ．なお線形リストの項表現 cons(lhs:E1, rhs:cons(lhs:E2, rhs:...)) に対して糖衣構文<E1, E2, ...>が用意されている．

child sort に含まれるグラフは集合を表している．集合は set 頂点を根とするグラフで表現する．属する要素は，その根からの elem 接続によって保持される．なお，集合の項表現 set(elem:E1, ..., elem:En) に対して糖衣構文 set<E1, ..., En>が用意されている．

3.3 書換え規則

REGREL+ は書換え規則を用いてグラフ書換えを行う．図 3 における Rule の定義のように，書換え規則もまたグラフによって表現されるのが特徴である．

書換え規則は規則左辺，述部そして規則右辺からなる．規則左辺はパターンと呼ばれるグラフによって書換え対象を表現する．パターンにマッチし，この書換えの候補となる部分グラフのことを redex と呼ぶ．述部は redex を実際に書換えるかどうかの判定を行うために用いられる．規則右辺は redex がどのように書換えられるかを表したグラフである．基本的には規則右辺の深い複製を作成し、それを redex の替わりとして置き換えることで書換えを行う．この複製元となるグラフのことをコンストラクタと呼ぶ．

書換え規則は->(lhs:Gl, rhs:Gr, pred:P1, ..., pred:Pn) という形式のグラフで表現される．ただし Gl, Gr, Pi はそれぞれ規則左辺，規則右辺そして述部であり，それぞれ省略可能である．また，糖衣構文として Gl, P1, ..., Pn -> Gr が認められている．

述部は Op(lhs:Pl, rhs:Pr) という形式のグラフで表現される．ただし Op, Pl, Pr はそれぞれ述語，規則左辺そして規則右辺である．述語には , !~, ~~, !~~, =>, !=>, =, !=を用いる事ができる．

3.4 パターンとパターンマッチ

パターンマッチでは部分グラフをパターンと見立て，対象の部分グラフがマッチするか判定を行う．定義の詳細は文献 [1] に譲り，その概要を以下に示す．

まず，パターンを表す頂点 v_P に頂点 v がマッチするとは，以下のどちらかを満たす場合をいう

- $label(v_P) = *$
- $label(v_P) = label(v)$

次に，パターンを表すグラフ G_P が，ある部分グラフ G にマッチするとは以下を満たす場合をいう．

- 全射 $f_V : V(G_P) \to V(G)$ が定義可能
- 全射 $f_E : E(G_P) \to E(G)$ が定義可能

ただし，$\forall v_P \in V(G_P)$ に関して v_P と $f_V(v_P)$ がマッチする．また，$\forall e_P \in E(G_P)$, に関して $f_V(src(e_P)) = src(f_E(e_P)), f_V(dst(e_P)) = dst(f_E(e_P))$ かつ $attr(e_P)$ に $attr(f_E(e_P))$ がマッチする．ここで，$src(e), attr(e), dst(e)$ は接続 e に関してその始点，属性そして終点を表す．

3.5 グラフ構築

グラフ構築は、複製元のグラフを元に新たにグラフを作成する手続きである．グラフ構築では、入力としてコンストラクタ G_C と前提となるパターンマッチの結果，そしてグラフを構築する対象のグラフ G をとる．そして、出力として G に G_C の複製を構築した結果 G' を返す．ただし G' は G に破壊的変更（頂点，接続の追加削除など）を行って得られたグラフである．

以下にコンストラクタに含まれる頂点と接続毎にどのような構築が行われるか概要を示す．

表 1　述語が真になる条件

述語	真となる条件
=>	左辺の根付き連結グラフを書換えたグラフで右辺にマッチするものが存在する
~	左辺の根付き連結グラフに右辺のパターンがマッチし，$v_L = f_V(v_R)$ である．ただし，v_L, v_R はそれぞれ左辺，右辺の根
~~	左辺の根付き連結グラフの部分グラフに，右辺のパターンがマッチする

- 頂点
  - ラベル: 同じラベルを持つ頂点を作成
  - 変数 V: f_V(V) を再配置
  - [V]: f_V(V) と同じラベルを持つ頂点を作成
- 接続
  - 接続 S(attr:D): 接続 (f_V(S), attr, f_V(D)) を作成
  - 削除辺 S(attr:/D), S(A:/D): 接続 (f_V(S), attr, f_V(D)), (f_V(S), f_V(A), f_V(D)) が存在するならば削除
  - 接続内包 S(A:D | A':D' in V): 頂点 f_V(V) を始点とする接続に関して，(f_V(V), A', D') にマッチするもの毎に f_V(S)(f_V(A):f_V(D)) を構築する

3.6 述語の評価

述語は書換え規則の述部の根である．REGREL+ での述語は 2 項演算子であり，それぞれ左辺と右辺に 1 つずつ根付き連結グラフをとる．表 1 には主要な述語と，それを評価した場合に真になるための条件を示す．なお，接頭辞!を伴う場合，その評価結果の否定を返す．

3.7 書換え

REGREL+ では書換え規則を用いてグラフの書換えを行う．

定義 3.4 書換え手続き

規則左辺 G_P，規則右辺 G_C を持つ書換え規則を用いて以下の手続きで書換えを行う

1. Matching: 書換え対象のグラフ G から，規則左辺 G_P にマッチする redex G_R を探索する．候補が複数存在する場合は，その中から非決定的に 1 つを選択する．
2. Authentication: 述部の評価を行い、全て真であるときのみ 3. 以降の手続きを行う．この際，述部をコンストラクタ，1. のパターンマッチの結果を用いて述部の構築を行い，述語の評価を行う．ただし，この構築，評価はグラフ G を書換えることなく行われる．
3. Building: G に規則右辺 G_C グラフ構築する
4. Redirection: redex G_R の根に関する接続を Building で構築された根に置き換える．具体的には G_R の根を始終点とする接続のうち，redex に含まれないものすべてを Building で得られた根に付け替える．

なお，規則左辺が省略された書換え規則では Building のみを行い，構築されたグラフの根が対象の sort から参照される．また，規則右辺が省略された規則ではまず Matching で redex を探索する．そして Authentication の結果が真ならば redex に含まれる頂点を全て削除する．削除された頂点への入出次接続は全て削除される．REGREL では頂点の直接的な削除

を行う方法は提供していなかったが，実用上行えると便利であるため REGREL+ ではその動作を認める．

3.8 書換え系の動作

グラフ書換え系としての REGREL+ の動作は，以下の手続きを繰り返し行うものとする．

1. Read: 入力の項表現に対応する sort 構造を系に追加する
2. Rewrite: 全ての sort が停止するまで書換えを行う
3. Print: 名前 0 を持つ sort に含まれるグラフを計算結果として表示する

Read では書換え系に対して sort の階層構造を追加する．この時，グラフ (項表現の EBNF における Graph) のみを追加した場合は暗黙に名前 0 を持つ sort のグラフとして追加される．

Rewrite では、親の sort に含まれるグラフのうち，書換え規則の条件を満たす部分グラフ全てを書換え規則として使用する．ただし例外として，書換え規則の一部に含まれる書換え規則は使用しない．

書換え可能な sort が複数存在する場合はそれらは並行に書換えが行われる．また，書換えの際に redex とそれを書換える規則の組が複数得られる場合がある．このとき，得られた redex と書換え規則の組の集合から，「安全な書換え」を行う事ができる redex の集合を非決定的に選択し，同時並行に書換えを行う．ここで「安全な書換え」とは，書換え規則 R_1, R_2 とそれぞれの redex G_{R_1}, G_{R_2} に関して，どちらの書換えを先に行っても両者の書換え結果が正しく得られる場合をいう．その厳密な定義は [1] に譲る．

例 3.2 線形リストの反転

図 5 は線形リストの反転を行う書換え規則と，その適用例である．ただし，>>>以下はは処理系への項表現の入力，---以下は書換え途中の名前 0 を持つ sort のグラフの出力，そして... 以下は書換えが停止した際の名前 0 を持つ sort のグラフの出力である．また，規則左辺における_頂点は，ラベル*を持つユニークな頂点の糖衣構文である．

```
>>> {R[reverse], R !~ _(result:_) ->
       R(result:nil),
     reverse(op:nil, result:L) -> L,
     reverse(op:cons(lhs:H, rhs:T), result:Lr) ->
       reverse(op:T, result:cons(lhs:H, rhs:Lr)),
     #0{ reverse(op:<1, 2, 3>) } };
--- reverse(op:<1, 2, 3>, result:nil);
--- reverse(op:<2, 3>, result:<1>);
--- reverse(op:<3>, result:<2, 1>);
--- reverse(op:nil, result:<3, 2, 1>);
... <3, 2, 1>
```

図 5 線形リストの反転

3.8.1 otherwise 規則

otherwise 規則は他の書換え規則が適用できなかった場合にのみ適用可能な，特別な書換え規則である．otherwise 規則は，->の代わりに|->を根に持つ書換え規則として表現される．

3.8.2 Garbage Collection

書換えの結果「有効でない」頂点は Garbage Collection によって系全体を表すグラフ G から削除される．ここで，有効な頂点は $V_v(G) = \{v | v \in rgraph_G(v_r), v_r \in Roots_G(s), s \in S(G)\}$，有効でない頂点は $V(G) - V_v(G)$ で定義される．

表 2 actor, message の表現

アクターモデルの要素	表現方法
actor の識別子	頂点
actor の種類	頂点が持つラベル
既知の actor	頂点への接続
message	頂点
actor A へ message M の送信	接続 (A, _*rcv*, M)
actor A が message M を受信	接続 (A, _*rcv*, M)

3.9 アクターモデル

アクターモデルは REGREL+ でもっとも簡潔に表現が行える計算モデルの一つであり，message 駆動型の動作を表現する方法として用いられる．アクターモデルでは，actor と呼ばれる計算主体が互いに message を送信し合い，受信した message に応じた action を行う事で計算を進めていく．アクターモデルの定義は [5] に準じるものとする．

REGREL+ では表 2 のようにアクターモデルを表現する．受信した message に対する action は根 A(_*rcv*:M) に必要な頂点と接続を追加したパターンを持つ書換え規則で表す事ができる．このアクターモデルの表現では，明示的に切断しない限り，受信した message への接続を維持する点に気をつけなければならない．よって，受信した message に関する動作を 1 回だけ行いたい場合は，動作を表す書換えと同時に，受信した message への接続を削除しなければならない．今回は簡単のため，送信後直ちに message が受信されるものとして表現している．ただし，action が行われるタイミングは非決定的である．

なお，message の送受信を表す項表現 `A(_snd:M)`, `A(_rcv:M)` そして `A(_rcv:/M)` に関して，それぞれ糖衣構文 `A ! M`, `A ? M` そして `A ?/ M` を認める．これらの糖衣構文に関して，`(A ! M)` のように括弧で括った場合，actor の根を表すものとする．

4 sort 操作

3 章で定義した REGREL+ では，書換え規則は Rule ::= Root [,PredList] -> Root とされている．つまり，規則右辺に sort を記述することは認められず，書換えによって sort の操作を行う事はできない．この事に関して書換えによる sort 操作を表現するために，Rule を Rule ::= Root [,PredList] -> Sort のように定義する方法も考えられる．今回，そのような定義を取らなかった理由としては，高階書換えを考慮すると規則左辺にも Sort を記述できるようにする必要があるからである．その場合，書換え規則によって sort に含まれる sort を書換えることができてしまう．そのような構造下では sort の階層構造がどの sort の書換え規則で操作されているのかが分かりづらく，書換えの順序を把握しにくいものと考えられる．

そこで，今回は sort 構造もグラフによって表現することで対応することを考える．

定義 4.1 sort のグラフ表現

- 名前 $n \in L$ を持つ sort は頂点 n によって表現する
- sort $s_p, s_c \in S(G) \subseteq V(G)$ に関して，s_p が s_c の親であるならば $(s_p, \_cld, s_c) \in E(G)$
- $s \in S(G), v_r \in roots(s)$ ならば $(s, \_ref, v_r)$.

このような定義を採った場合にも，任意の sort から他の sort が参照できてしまうのは好ましく無い．できる限り，sort の操作を受け持つ sort のみが他の sort を参照することができるようにし，それ以外の sort は sort に含まれるグラフ構造の

表 3　meta sort を構成する sort

sort (sort 名)	親	子
	グラフ	
	備考	
top sort (_top)	無し	entirety sort
	normal sort を書換える書換え規則	
entirety sort (_ent)	top sort	boundary sort
	boundary sort	
boundary sort (_bnd)	entirety sort	親を持たない normal sort
	グラフを含まない	
	normal sort との窓口を持つ	

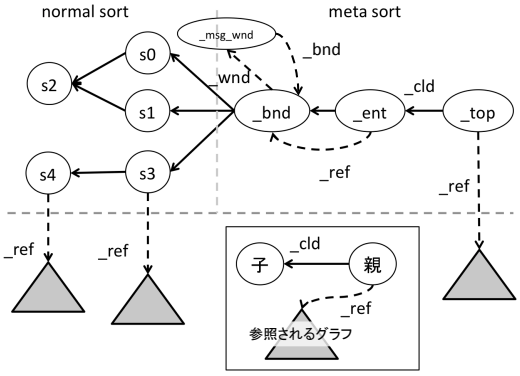

図 6　meta sort を伴う sort 構造

みを参照するようにしたい．そのため，次のような sort 構造のもとで計算を行う．

定義 4.2 meta sort を伴う sort 構造

sort の操作を行うための sort を meta sort と呼ぶ．また meta sort で無い sort を区別して normal sort と呼ぶことにする．meta sort は top sort, entirety sort そして boundary sort の 3 つの sort で構成される．それらの親子関係や参照するグラフを表 3 に記す．

この sort 構造の元では，最初に entirety sort が書換え可能になる．entirety sort は boundary sort を根とするグラフを含むため，normal sort の sort 構造以下のグラフが top sort に含まれる書き換え規則によって書換えられる．この時，書換え対象には normal sort を表現する頂点と，親子関係を表す *_cld* 接続からなる部分グラフも対象となるため，normal sort での sort の階層構造を書換えることができる．

normal sort の構造を meta sort から参照することが可能になったので，normal sort 側から meta sort に sort 操作を要求する方法について検討する．normal sort が含むグラフを参照して，top sort に含まれる書換え規則が sort 操作を行う方法を採ることも可能であるが，探索の範囲が広がるため探索効率が悪い．可能ならば，normal sort が含むグラフを参照することなく処理を行いたい．そのため，normal sort から窓口とよばれる頂点へ向けて meta sort へ sort 操作要求の message を送信し，一元管理する．窓口は normal sort，meta sort の双方から参照可能な特別な頂点であり，`_msg_wnd` 頂点で表され，`_bnd` 頂点との接続 (`_bnd`, *_wnd*, `_msg_wnd`) を持つ．また，窓口は全ての normal sort への`_srt` 接続を持っている．なお，項表現では窓口を特別な変数`$Window` によって表す．

表 4　sort 操作 message

`create(name:N, graph:G)` sort を作成し，グラフを追加する	
`N`	作成する sort 名
`G`	追加するグラフ．任意個数指定可
`connect(parent:Np, child:Nc)` sort に親子付けする	
`Np`	親の sort 名
`Nc`	子の sort 名
`remove(name:N)` sort の削除を行う	
`N`	削除する sort 名
`add_graph(name:N, graph:G)` sort にグラフを追加する	
`N`	追加する sort の名前
`G`	追加するグラフ

```
C[create](name:N) -> N & ($Window !
    create(name:N, graph:G | graph:G in C));
```

図 7　normal sort から meta sort への要求送信の例

例えば図 7 の書換え規則によって normal sort から meta sort へ窓口を通じて sort 操作の要求を送信することができる．なお，この書換え規則は Rule ::= Graph -> Graph_M & Graph_S の形式で記述されており，Building の際に $\text{Graph}_\text{M}, \text{Graph}_\text{S}$ の両方を構築する．ただし，Redirection の対象となるのは Graph_M だけである．また書換え規則を表すグラフにおいては，->頂点から Graph_S の根への `sub` 接続によって表現する．

窓口が message を受信した事によって entirety sort が書換え可能になる．図 8 の書換え規則は，送信された `create` message に対する action であり，新規の normal sort を作成する．

```
W[_msg_wnd](_bnd:Sb[_bnd]) ? M[create](name:N),
  W !~ _(_srt:[N])
  -> W(_srt:Sn[N](_ref:G | graph:G in M)) ?/
        M & Sb(_cld:Sn);
```

図 8　create message に対する action

sort を操作するための代表的な message を表 4 に記した．また，これらの実装の一部を付録 A に記した．これらの message では，パラメータとして sort を直接扱うのではなく，対象となる sort の名前を用いて行う．なお，窓口へ有効で無い message が送信された場合，top sort の otherwise 規則によって全て破棄される．entirety sort が停止した後は boundary sort が書換え可能になるが，グラフを含まないので entirety sort に含まれる書換え規則はいかなるグラフにも適用されない．また boundary sort が停止すると，その子である normal sort が書換え可能になるが boundary sort は書換え規則を含まないのでいかなるグラフも書換えられない．このように boundary sort は meta sort と normal sort の境界としての役割も持っている．

5　sort を用いたプログラム手法

5.1　部分グラフの Lock

sort 操作の応用例として，ある部分グラフに lock をかけて特定の書換え規則を優先的に適用させる sort を作成する例を示す．そのような動作は，文献 [1] での定義における REGREL では，記述が困難であった．REGREL+ では新し

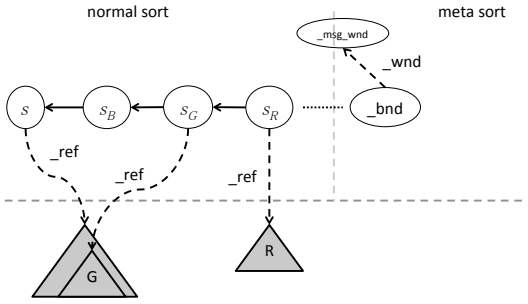

図 9　lock 処理を行う sort の階層構造

く lock 空間用の sort を作成して，そこで優先的に書換えを行うことで lock を表現できる．

図 10 は lock(graph:G, rules:R, sort:N) message を窓口へ送信することで，グラフ G の lock 要求を行う書換え規則である．

```
W[_msg_wnd] ? M[lock](graph:G, rules:R, sort:N)
  -> W ?/ M &
       (W ! create(name:Sr[], graph:R, graph:
              (|-> & (W ! remove(name:Sr),
                      W ! remove(name:Sg),
                      W ! remove(name:Sb)))),
        W ! create(name:Sg[], graph:G),
        W ! create(name:Sb[]),
        W ! connect(parent:Sr, child:Sg),
        W ! connect(parent:Sg, child:Sb),
        W ! connect(parent:Sb, child:N))
```

図 10　lock message に対する action

message が送信され，図 10 の書換え規則が適用されると 3 つの sort s_R, s_G, s_B が作成される．lock message を送信した sort を s とすると，親子関係は s_R, s_G, s_B, s の順に親から子の関係が追加される．そのため，書換えでは s_R のグラフ R に含まれる書換え規則が s_G のグラフ R に適用される．そして s_G が停止したのち，グラフを含まない s_B は直ちに停止する．最後に s の全ての親が停止したら s の書換えが行われる．よって，s の書換えが起こるよりも先に s_G の書換えが行われる事がわかる．s_G と s の間に s_B を挟むのは，s_G に含まれる書換え規則が s のグラフに適用されるの防ぐためである．

図 10 の記述では窓口への message 送信が羅列され煩雑であるため，次の糖衣構文を導入する．高表現での書換え規則規則右辺において，&以下に sort の階層構造が記述されている場合，対応する sort を構成するための create, connect message に置き換えられるものとする．図 11 の書換え規則は図 10 を糖衣構文を用いて書きなおしたものである．

```
W[_msg_wnd] ? M[lock](graph:G, rules:R, sort:N)
  -> W ?/ M & {R, (|->  & (W ! remove(name:Sr),
                           W ! remove(name:Sg),
                           W ! remove(name:Sb))),
                {G, { #N } } };
```

図 11　図 10 の糖衣構文を用いた書換え

図 12 は normal sort から meta sort へ lock message を送信する規則の例である．この書換え規則では，lock message の引数として lock 要求を行った sort 名を与えるために，特

```
lock(graph:G, rules:R) ->
   G & $Window ! lock(graph:G, rules:R,
                      sort:$Current)
```

図 12　lock 要求を送信する規則例

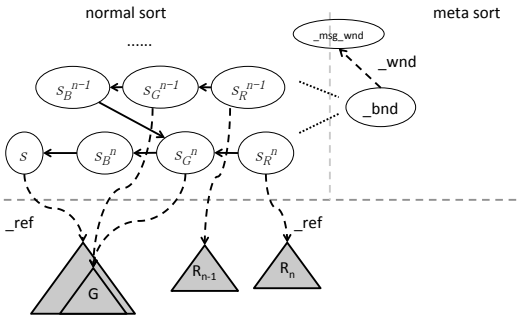

図 13　逐次処理を行う sort の階層構造

別な変数$Current を使用している．これは現在書換えの対象となっている sort の名前を表すものである．sort を表す頂点そのものではなく，同じラベルを持つ異なる頂点が構築されるものとする．

図 12sort では書換え規則|-> & remove ... という形式の書換え規則が使用されている．これは s_G に対して通常の書換え規則の適用が行えなくなった後に適用され，sort s_R, s_G, s_B の削除要求を meta sort へ送信する．規則左辺が空であるため，書換え規則群 R を適用できなくなった場合に必ず適用される．そして書換えが適用されると，直ちに meta sort へ書換えが遷移し，sort s_R, s_G, s_B は削除されるため二度目以降の適用は起き得ない．なお，この otherwise 規則が存在するため，R に otherwise 規則を含めてはならない．

例 5.1 集合からの要素の削除

集合から条件を満たす要素を削除する規則を定義する．

```
remove_elem(set:S, preds:P) ->
  lock(graph:S, rules:set<P,
       (set(elem:E), check(op:E) => true ->
    set(elem:/E))>)
```

図 14　集合からの要素の削除

図 14 の規則は対象となる集合 S の要素 E に関して，P に含まれる書換え規則によって check(op:E) を true に書換えられるのであれば，その要素 E を S から削除する．この処理は S を lock して行うので，remove_elem の書換えを行った sort からは削除途中の中間状態を観測されることがない．

5.1.1　逐次処理

lock 手法の拡張として，部分グラフ G に対して書換え規則群の列 $R_1, \ldots, Rn$ $(n > 1)$ の要素をそれぞれ順に，適用できなくなるまで適用する逐次処理の表現方法について論じる．

seq_process(graph:G, rules_seq:S, sort:N) message はグラフ G を lock し，線形リスト S[cons](lhs:Rn, rhs:...cons(lhs:R1, rhs:nil)) の要素である書換え規則群 $R_1, \ldots, R_n$ をそれぞれ停止するまで適用する．ただし，S には $R_1, \ldots, R_n$ を逆順に並べる．

動作内容は lock とほぼ同様である．R_n に対して sort s_R^n, s_G^n, s_B^n を作成した後，s_R^n に R_n を追加し，s_G^n に G を追加

する．そして，s_G^n を対象の sort として再帰的に seq_process message を送信していく．逐次処理要求の送信とそれに対する action に対応する書換え規則を図 15 に示した．

```
seq_process(graph:G, rules_seq:S)
  -> G & $Window ! seq_process(graph:G,
                               rules_seq:R,
                               sort:$Current);
W[_msg_wnd] ?
    M[seq_process](graph:G, sort:N,
                   rules_seq:cons(lhs:R, rhs:T))
  -> W ?/ M &
       ({ R, (|-> & (W ! remove(name:Sr),
                     W ! remove(name:Sg),
                     W ! remove(name:Sb))),
          {G, { #N } } },
        W ! seq_process(graph:G, sort:Sg,
                        rules_seq:T));
```

図 15　逐次処理要求とその action

例 5.2 トポロジカルソート

seq_process message を用いて，有向グラフのトポロジカルソートを行う例を示す．ts(roots:S) は，有向グラフの全頂点の集合 S をトポロジカルソートして得られる線形リストを返す．ただし，有向グラフは頂点を vertex 頂点，接続を edge 接続で表す．また，処理の過程で対象のグラフの edge 接続は全て削除される．なお閉路を含むグラフが与えられた場合 error を返す．

```
Ts[ts](roots:set) ->
  seq_process(graph:Ts(list:nil),
              rules:<R3, R2, R1>) &
    (R1[set]<
       Ts1[ts](roots:S1[set](elem:V1[vertex]),
                             list:T1),
           V1 !~ _(edge:_),
           V1 !~ (_ ? checked) ->
         Ts1(list:cons(lhs:V1 ? checked,
                       rhs:T1)),     // 規則 1-1
       Vs[vertex](edge:Vd[vertex] ? checked) ->
         Vs(edge:/Vd)>,              // 規則 1-2
     R2[set]<
       Ts2[ts](roots:S2[set](elem:_),
               list:L2) -> error,    // 規則 2-1
       Ts3[ts](roots:S3, list:L3),
           S3 !~ _(elem:_) -> L3,    // 規則 2-2
       V2 ? M[checked] -> V2 ?/ M>); // 規則 2-3
```

図 16　トポロジカルソート

処理の概要を以下に示す．

- 規則 1 群

規則 1-1　集合に含まれる出次辺を持たない vertex 頂点を集合から除外し，線形リストの先頭に追加する．そしてその頂点に checked message を送信する

規則 1-2　checked message を受信した vertex への edge 接続を削除する

- 規則 2

規則 2-1　集合に要素が残っている場合，ソートは失敗であるので error に書換える

規則 2-2　集合に要素が残っていない場合，ソートは成功したので結果の線形リストに書換える

規則 2-3　checked message を受信している頂点からその message を切断する

表 5　REGREL+ におけるアスペクトの表現

機能	REGREL+ での表現
advice	高階書換え規則
join point	ある sorts_P に含まれるグラフ
pointcut	ある sorts_P とその親の sorts_A
weaving	join point の高階書換え

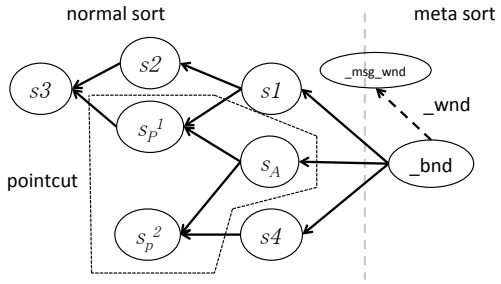

図 17　pointcut の表現

5.2　アスペクト指向プログラミング

アスペクト指向プログラミング (AOP)[6] の表現方法を示す．プログラム中には join point と呼ばれる，処理を割り込ませる事が可能な位置が用意されている．割り込ませる処理の事を advice と呼び，割り込ませる行為を weaving と呼ぶ．また，同じ性質を持つ join point の集合を pointcut と呼ぶ．pointcut に含まれる全ての join point に対して同時に advice を weaving する事が可能である．まず AOP の各要素と，REGREL+ での表現方法を表 5 に示す．また図 17 は pointcut の表現例である．

weaving は sort s_A に advice となる高階書換え規則を追加することで，sort s_P に含まれる join point を表すグラフや書換え規則を高階書換えすることで行う．これらの sort s_P, s_A は pointcut を形成し，sort s_A は advisory sort と呼ばれる．advisory sort は複数の子を持つことが可能なので，横断的に advice を weave することが可能である．

例 5.3 計算過程のログ保存

図 18 は書換え規則を特殊化して，書換えが行われた sort 名，適用された書換え規則，そして書換え結果を纏めた log message を窓口に送信する advice である．この advice を任意の advisory sort に入れることで，その pointcut に含まれる書換え規則は全て，書換えが適用される毎に log message を送信するように特殊化される．log message を受信した後，log を構成するための書換え規則については割愛する．

```
R[->](lhs:LHS, rhs:RHS),
    R !~ _(sub:$Window ! log) ->
  (LHS -> RHS &
     $Window ! log(graph:RHS,
                   rule:R, sort:$Current)));
```

図 18　計算過程のログを取る advice

5.3　部分評価

本節では書換え規則の規則右辺を部分的に高階書換えすることで，コンストラクタの一部を部分評価する方法について論じる．単に通常の書換え規則を用いて部分評価を行お

うとすると，規則左辺を書換えた場合にパターンの意味が変更されてしまい，等価な書換え規則にならない場合がある．そのため書換え規則 LHS -> RHS が規則右辺にのみ適用されるように，この書換え規則を元に高階書換え規則 R(_ -> RHS')，RHS' ~~ LHS -> lock(graph:R，rules:(LHS -> RHS)) を作成して適用する．この書換え規則は任意の書換え規則にマッチするが，redex の持つ規則右辺 RHS' の部分グラフがパターン LHS にマッチするときのみ書換えが行われる．

部分評価を行うための対象となる書換え規則は normal sort に遍在している場合がある．それら全てに部分評価を行うために，自分以外の全ての normal sort の親である ansector sort を作成する．この ansector sort(_ans) は， normal sort の ansector sort 以外に存在する join point を含む pointcut の advisory sort とみなすことができる．

図 19 は書換え規則 LHS -> RHS から特殊化用の書換え規則を作成し，advisory sort へ追加する pe message に対する action である．

```
W[_msg_wnd] ? M[pe](rule:(LHS -> RHS)) ->
  W ?/ M & ($Window !
    add_graph(name:_ans, graph:(
      R[->](rhs:RHS')，RHS' ~~ LHS ->
        (R & lock(graph:RHS',
                  rules:(LHS -> RHS))))));
```

図 19　部分評価用の message に対する action

6　考察

6.1　計算順序の制御

書換え系では一般的にどの書換え規則が適用されるか非決定的である．そのため，ある順序に基づいて逐次的に書換え規則を適用するするためには，何らかの手段によって優先順位を与えなければならない．ここでは lock 処理や逐次処理の表現について REGREL, LMNtal と比較する．

REGREL では lock 動作は記述自体が困難である．また，ある階層内においては分類を用いて逐次処理を行うことが可能であるが，そのステップ数を動的に変更するような動作は難しい．そして，その動作中に書換え対象のグラフの中間状態を他の分類から参照することが可能である．LMNtal では新たな膜を生成し，その中にグラフと書換え規則を追加するだけで lock が表現できる．そして膜を入れ子にするだけで逐次処理も記述することができる．REGREL+ でも同じような手法で同様の表現が可能であるが，膜を使った手法よりも多くの sort が必要となる．

6.2　計算順序制御構造の操作方法

REGREL では階層構造への新たな階層の追加や，分岐を作成することは認められていない．そのため，一度決定した計算順序を動的に変更する事は難しい．LMNtal では対象としている膜の操作や，その内部に新規の膜を生成することなどが可能である．ただし，それ以外の祖先や兄弟などの膜に対する操作を行う事はできない．一方，REGREL+ では meta sort から見た normal sort は単なるグラフでしか無いため，書換え規則を用いてその構造を変更する事が可能である．このことから，制御構造の操作に関する柔軟性に関しては既存の手法よりも優れていると言える．

6.3　アスペクト指向プログラミングの表現

ここでは，グラフ書換え系によって AOP の表現する Aßmann らの手法 [7]，REGREL での手法と REGREL+ での提案手法との比較を行う．

表 5 に記した様に，join point を書換え規則やグラフで，advice を高階書換え規則で表現し，高階書換えで weaving を表現する点は同様である．しかし，Aßmann らの手法では pointcut は扱われていない．一方，REGREL での表現では，分類を用いることで pointcut の表現を行うことが可能である．ただし， join point が複数の階層に跨って存在する場合にはそれぞれの階層に適用するように複数の advice を対応する階層に追加しなければならない場合がある．それ対して提案する REGREL+ の手法では，単一の advisory sort に advice を追加するだけで，複数の join point を特殊化することが可能であり，より pointcut の性質を詳細に表現できているといえる．

7　おわりに

計算順序制御のための構造として sort を提案した．そして sort の階層構造のもとで定義されたグラフ書換え言語 REGREL+ の定義とプログラム例を示した．LMNtal の膜に似た制御方法での lock や逐次処理を行う事や，REGREL と同様に上位から高階書換えの影響を伝搬していくことが可能であることを示した．また既存手法では表現しにくい系を横断する weaving や，部分評価が表現可能であることを示し，既存の計算順序制御機構に対する優位性を示した．

参考文献

[1] 東達軌, 武田正之．“グラフ書換え言語 regrel における並行グラフ操作クエリの表現”．*FIT2010*, (2010).

[2] 乾敦行, 工藤晋太郎, 原耕司, 水野謙, 加藤紀夫, 上田和紀．“階層グラフ書換えモデルに基づく統合プログラミング言語 LMNtal”．コンピューターソフトウェア, Vol. 25, No. 1, pp. 124–150, (2008).

[3] 上田和紀, 加藤紀夫．“言語モデル LMNtal”．コンピューターソフトウェア, Vol. 21, No. 2, (2004).

[4] Gheorghe Păun．“introduction to membrane computing”．In Gabriel Ciobanu, Gheorghe Pun, and Mario J. Prez-Jimnez, editors, *Applications of Membrane Computing*, Natural Computing Series, pp. 1–42. Springer Berlin Heidelberg, (2006).

[5] 所真理雄, 松岡聡, 垂水活幸．“オブジェクト指向コンピューティング”．岩波書店, (1993).

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. “aspect-oriented programming”. In *ECOOP*. SpringerVerlag, (1997).

[7] Uwe Aßmann and Andreas Ludwig. “aspect weaving with graph rewriting”. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, GCSE '99, pp. 24–36, London, UK, (2000). Springer-Verlag.

付録 A　sort 操作を行う書換え規則

```
// connect
W[_msg_wnd](_srt:Sp, _srt:Sc) ?
    M[connect](parent:Np, child:Nc),
    Sp ~ Np, Sc ~ Nc -> W ?/ M & Sp(_cld:Sc);

// add_graph
W[_msg_wnd](_srt:S) ?
    M[add_graph](name:N, graph:G),
    S ~ N, -> W ?/ M & S(_ref:G);

// remove
W[_msg_wnd] ? M[remove](name:N),
  W ~ _(_srt:[N]) -> Sb ?/ M & Sn ! remove_sort;
S ? remove_sort -> ;
```