A-021

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
Concurrent graph query language in graph rewriting language REGREL

東 達軌† Tatsuki Higashi
武田 正之‡ Masayuki Takeda

1 はじめに

近年では計算の対象として挙げられる問題は多様化してきている．それらの問題を表現するための構文や意味論も多様であり，既存のプログラミング言語を用いて表現，計算を行うことが困難である場合がある．その問題を解決するためには，様々な問題領域の構文や意味論をユーザが任意に定義可能であるような環境が望まれる．

本研究では，そのような汎用のメタプログラミングが可能である環境の構築を目的としている．最初の段階として，各種構文や意味の表現が可能な計算モデルの構築を目標とする．

本論文では，そのような計算モデルとして，リフレクティブなグラフ書換え言語 REGREL(REflective Graph REwriting Language) を提案する．REGREL は DACTL [1] を基礎として定義されているが，より表現力を高めるための工夫が行われている．そして，REGREL を用いた応用として，汎用のグラフ操作クエリ言語を定義してその評価を行う．

REGREL はグラフ書換え系 [2, 3, 4] であり，REGREL グラフと呼ばれる，頂点と接続の両方にラベルを持つ有向グラフを書換え対象とする．REGREL グラフの書換えは書換え規則によって定義される．その書換えの対象はパターンマッチで探索され，さらに述語を用いて絞り込むことができる．また，分類と呼ばれる機構によって書換え規則の適用範囲を限定することができる．

REGREL では REGREL グラフによってプログラムの構造や構文を表現し，書換え規則によってその意味を表現する．そして，書換え規則もまたグラフで表現されているため，書換え規則を書換える高階書換えが可能となっている．その特徴を利用して，リフレクションを実現している．また，REGREL の書換えは並行かつ非決定的に行われる．なお逐次動作や決定性動作はその特別な場合として表現される．

XQuery[5] や LINQ[6] などの木構造やグラフ操作クエリは，基本的には逐次的に動作するものとして定義されている．それに対して，必要な箇所では明示せずとも逐次処理や同期を行い，基本的には並行に動作するクエリが定義できれば，それを用いた計算は並行計算資源を有効に活用したものになると期待される．今回は REGREL を用いた応用として，そのような汎用なグラフ操作クエリを REGREL 上に定義する．なお，このクエリはグラフ書換えとアクターモデル [7] に基づいて定義されている．

本論文ではまず REGREL の定義を示す．次に応用として REGREL の上に汎用グラフ操作クエリを定義し，その実例を示す．最後に，既存のグラフ書換え系である DACTL [1] そして LMNtal [8, 9] との比較を行う．

2 グラフ書換え言語 REGREL

2.1 REGREL グラフ

REGREL において書換えの対象となるグラフはラベル付き頂点とラベル付き有向辺からなる有向グラフであり，REGREL グラフと呼ばれる．以下，そのラベル付き有向辺を誤解の無い範囲で単に接続と呼ぶ．

ここで，グラフの形式的定義を与える．ラベルの集合 L の下で，グラフ G に含まれる頂点の集合を $nodes(G)$ で表す．なお，頂点 $n \in nodes(G)$ の持つラベルを $label(n)$ で表し，ラベル l を持つ頂点を l 頂点と呼ぶことにする．また，接続の集合を $arcs(G)$ で表す．ただし，$(n_s, n_a, n_d) \in arcs(G)$,$(n_s, n_a, n_d) \in nodes(G)$ である．これは，始点 n_s から終点 n_d へ，ラベル $label(n_a)$ をもつ接続が存在することを表している．この様に，接続は頂点の 3 つ組で表され，2 つ目の頂点 n_a が持つラベルによって接続の持つラベルを表現する．そして接続 $a \in arcs(G)$ の持つラベルを $attr(a)$ で表し，接続のもつラベルのことを特に属性と呼ぶことにする．属性 α を持つ接続を α 接続と呼ぶことにする．なお，同一の始終点を持つ任意の接続 $a, a' \in arcs(G)$ に関して，$attr(a) \neq attr(a')$ であってはならない．

REGREL グラフにおける根付き連結グラフとは，根となる頂点 $n \in nodes(G)$ 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．以下根付き連結グラフを誤解の無い範囲で単にグラフと呼ぶ．なお，図 1 に簡単なグラフの例を 2 つ示した．

REGREL グラフを文字列表記で表すための項表現を図 2 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．また，終端記号 level, label, var, op に関しては，その字句表現を正規表現で記した．

REGREL での一般的なグラフ Graph は，根付き連結グラフの根 Root の列として表される．根は頂点 Node または書換え規則 Rule で表される．頂点 Node は分類 Catg の列，変数名，ラベル，接続 Arc の列で表される．頂点に変数名が与えられていない場合，与えられたラベルを持つ固有の頂点を表すとする．この場合，必ずラベルを持たなければならない．一方，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．この時，少なくとも一つの Node はラベルを持たねばならず，異なるラベルを持つものが存在してはならない．なお，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．

例 2.1 図 1 のグラフに対応する項表現を以下に示す．

```
A[a](foo:b(bar:c(baz:A)))
append(car:cons(car:1, cdr:N[nil]), cdr:N)
```

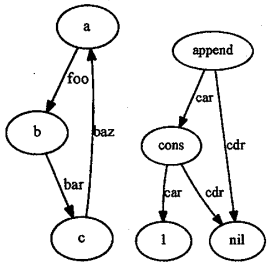

図 1 REGREL グラフの例

† 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

```
LGraph   ::= level CGraph
CGraph   ::= {CatgList} ( CGraph ) | Graph
Graph    ::= Root | (Root {, Root})
Root     ::= {CatgList} ( Root ) | Rule | Node
Node     ::= Name [[ Name ]] [(ArcList)]
CatgList ::= [Catg {, Catg}] [/ Catg {, Catg}]
Catg     ::= label
Name     ::= label | var
ArcList  ::= (Arc | Comp) {, (Arc | Comp)}
Arc      ::= Node(: | !:)Node
Comp     ::= Arc | Arc in Node
Rule     ::= Root [,PredList] -> Root | (Rule)
PredList ::= Pred {, Pred}
Pred     ::= Root op Root

level    ::= [0-9]+
label    ::= [_+\-\.a-z0-9][_+\-\.a-zA-Z0-9]
             | [\".*\"]
var      ::= [A-Z][a-zA-Z]*
op       ::= ~ |!~ |=>|!=>|==|!==
```

図 2　REGREL グラフの項表現

2.2　書換え規則

REGREL における書換え規則は以下の要素を持つ．

- 書換え対象のパターンを表現する規則左辺
- 書換え対象に対して絞り込みを行う述語
- 書換えの結果を表現する規則右辺

規則左辺，規則右辺はそれぞれを必ず 1 つだけ持つ．また，述語は有限個持つことができる．

先に述べた様に，REGREL では書換え規則もまたグラフによって表現される．書換え規則は根付き連結グラフで表現され，その根 n_R は必ず $label(n_R) =$ -> でなければならない．また，n_R を始点とする接続 a_L, a_R で $attr(a_L) = \_lhs, attr(a_R) = \_lhs$ を満たすものが必ず存在しなければならない．このとき a_L, a_R の終点をそれぞれ規則左辺，規則右辺の根とする．また，n_R を始点とする接続 a_P が $attr(a_P) = \_pred$ である場合，a_P の終点を述語の根とする．

規則右辺や述語には再配置頂点と呼ばれる特別な頂点が現れる事がある．この頂点は書換えの際にパターンにマッチした頂点が，書換え後どこに再配置されるかを表現するために用いられる．そのため，再配置される頂点を保持するための *_ref* 接続を持っている．再配置頂点の具体的な説明は，定義 2.3 で行う．

また，書換え規則は付加情報として削除辺と内包表記定義を持つ事ができる．削除辺は書換えの際に削除される接続を明示するものである．書換え規則の根 n_R に関して接続 $(n_R, n_{rmv}, n_s), (n_s, n_a, n_d)$，$label(n_{rmv}) = \_rmv$ が存在し，n_s, n_d が再配置頂点の時，接続 (n_s, n_a, n_d) を削除辺と呼ぶ．削除辺の具体的な説明は定義 2.5 で行う．

述語も書換え規則と同様に，左辺と右辺を 1 つずつ持つ根付き連結グラフとして表される．その根を n_P とするとき，$label(n_R)$ には演算子~，!~，=>，!=>，==，!==の 6 つが認められている．また，n_R を始点とする接続 a_L, a_R で $attr(a_L) = \_lhs, attr(a_R) = \_lhs$ を満たすものが必ず存在しなければならない．それぞれの演算子，規則左辺や規則右辺の意味は次節で説明する．

以下に書換え規則に関する項表現の概要を示す．書換え規則の項表現 Rule は規則左辺 (Root)，述語 (Pred) の列，->演算子，規則右辺 (Root) で表現される．規則左辺，規則右辺は Root であるので，書換え規則がパターンとして現れたり，書換え結果が書換え規則であってもよい．書換え規則において，そのどれもがラベルを持たない変数は暗黙にラベル*を持つものとする．さらに，その変数名が書換え規則の中で唯一で，規則左辺または述語~，!~の右辺に現れる場合は変数名に _ (アンダーバー) を用いてよい．また，ある変数が規則左辺と右辺にまたがって出現する場合，規則左辺では同じ頂点の出現を表すが，規則右辺ではその左辺に現れる頂点を参照する再配置頂点を表す．

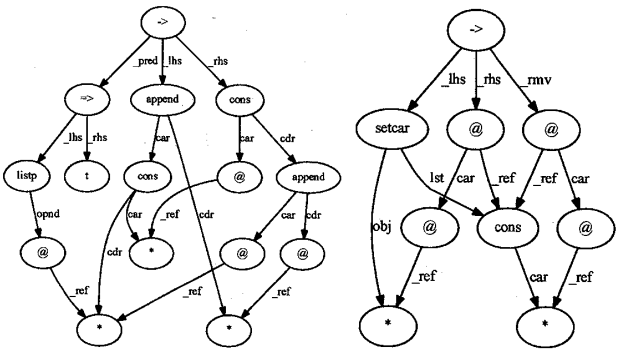

図 3　書換え規則の例

削除辺は，接続記号:ではなく!:によって表される接続として規則右辺に現れる．また，書換え規則の規則右辺では，ある頂点の持つ接続の集合を基とした新たな接続の作成を表す表記として，接続内包記法 Comp を認める．接続内包表記は Arc_1 | Arc_2 in Node の形式で記述され，接続内包定義と対応している．これは，Node の持つ Arc_2 にマッチする接続に関して，Arc_1 の接続を持つことを意味する．接続内包表記を含む書換え規則の動作は，定義 2.6 で示す．

例 2.2 図 3 の書換え規則に対応する項表現を以下に示す．

```
append(car:cons(car:E, cdr:L1), cdr:L2)),
  listp( op:L1 )
    -> cons(car:E, cdr:append(car:L1, cdr:L2))
setcar(lst:cons(car:Car), obj:E) ->
  cons(car!:Car, car:E)
```

2.3　書換え規則の適用

REGREL では書換え規則を用いてグラフの書換えを行う．書換え動作の定義を行う前に，以下の定義を行う．

定義 2.1 パターンマッチ
グラフ G_P に部分グラフ G_T がマッチするとは以下の場合をいう．

- G_P と G_T がグラフ同型
- 対応する頂点は同じラベルを持つ．ただし，G_P の頂点が*ラベルを持つならば G_T 側の頂点は任意のラベルを持っていてよい
- 対応する接続は同じ属性を持つ

パターンマッチにおいて，グラフ G_P はパターンと呼ばれる．また，パターン G_P に部分グラフ G_T がマッチするとき，G_P の頂点 n_P に対応する G_T の頂点 n_T の関係を $n_P \rightarrow_{(G_P,G_T)} n_T$ で表す．誤解の無い範囲で，$\rightarrow_{(G_P,G_T)}$ の略記として $\rightarrow$ を認める．

例 2.3 グラフ G_P `cons(car:L, cdr:R)` にグラフ G_T `cons(car:0, cdr:nil)` がマッチする．ただし，`cons` $\rightarrow$ `cons`，`L` $\rightarrow$ `0`，`R` $\rightarrow$ `nil` である．

またグラフ G'_P cons(car:L) にグラフ G_T がマッチする．ただし，cons → cons，L → 0 である．このパターンマッチでは，パターン G'_P は G_T の cons 頂点と，それを始点とする一部の接続にのみマッチしている．この様なパターンマッチを部分マッチと呼ぶ．

定義 2.2 述語演算子
REGREL には 6 種類の述語演算子が存在し，2 つの根付き連結グラフをとる．述語が真になるための条件を以下に示す．

- =>: 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場合のみ真．
  なお糖衣構文として，Root => true を単に Root と記述する事を認める．
- !=>: 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しない場合のみ真
- ~: 左辺の根付き連結グラフに，右辺のパターンがマッチする場合のみ真
- !~: 左辺の根付き連結グラフに，右辺のパターンがマッチしない場合のみ真
- ==: 左辺の根と右辺の根が同一の頂点である場合のみ真
- !==: 左辺の根と右辺の根が同一の頂点でない場合のみ真

定義 2.3 再配置頂点
グラフ G の頂点 n が@頂点であり，$(n, n_a, n_r) \in Arcs(G)$，$label(n_a) = \_ref$ が存在する場合，頂点 n を再配置頂点と呼ぶ．また n_r を n の参照頂点と呼び，$ref_G(n)$ で表す．そして (n, n_a, n_r) を参照接続とよぶ．

定義 2.4 グラフ構築
パターンを表すグラフ G_P に，グラフ G の部分グラフ G_T がマッチするとする．そのとき，グラフ G にグラフ G_C を構築する手続きを以下に定義する．

1. 参照頂点でない頂点 $n \in nodes(G_C)$ に関して，頂点 n' を G に追加する．ただし，$label(n) = label(n')$，$n \rightarrow_{(G_C,G)} n'$ とする
2. 参照頂点である頂点 $n \in nodes(G_C)$ に関して，$n \rightarrow_{(G_C,G)} n'$ とする．ただし，$ref_{G_C}(n) \rightarrow_{(G_P,G_T)} n'$ である
3. 非参照接続 $(n_s, n_a, n_d) \in arcs(G_C)$ に関して，接続 (n'_s, n'_a, n'_d) を G に追加する．ただし，$n_s \rightarrow_{(G_C,G))} n'_s$，$n_d \rightarrow_{(G_C,G))} n'_d$，$label(n_a) = label(n'_a)$ である

例 2.4 グラフ G_P a(foo:F) に，グラフ G a(foo:b(bar:c)) の部分グラフ G_T a(foo:b) がマッチしたとする．この時グラフ G に G_C d(baz:F) を構築する．（厳密には グラフ G_R (a(foo:F)，d(baz:@(_ref:F)) に関して，a を根とする根付き連結グラフを G_P，d を根とする根付き連結グラフを G_C とする）

1. d(baz:@(_ref:F)) の d は通常頂点であるので，G にそれを追加する．

   (a(foo:b(bar:c)), d)

2. d(baz:R[@](_ref:F)) の R は参照頂点である．R →$_{(G_C,G)}$ b とする．
3. 接続 (d, baz, R) ∈ G_C に関して，接続 (d, baz, b) を G に追加する．

   (a(foo:F[b](bar:c)), d(baz:F))

定義 2.1, 2.2, 2.3, 2.4 を用いて書換え手続きを定義する．なお，この手続きは DACTL の書換えを基礎として定義されている．

定義 2.5 書換え手続き
書換えの手続きを以下の様に定義する．

1. Matching: 書換え対象のグラフから，規則左辺にマッチする部分グラフ (redex) を発見する．マッチする箇所が複数存在する場合は，非決定的に 1 つを返す
2. Authentication: redex の書換えを行うべきか判定する．述部の根の集合を $n_{p_1}, \ldots, n_{p_n}$ とする．まず，$1 \leq i \leq n$ に関してそれぞれ $nodes(G_i) = nodes(G), arcs(G_i) = arcs(G)$ である G_i を作成し，そのグラフに対して n_{p_i} を根とする根付き連結グラフ構築する．構築された述部の書換え結果が真でないならば，その redex の書換えは行わない．
3. Building: 頂点，接続の追加，接続を削除する
   G に規則右辺 G_C を構築する．また，削除辺 (n_s, n_a, n_d) が存在する場合は (n'_s, n'_a, n'_d) を削除する．ただし，$ref_{G_C}(n_s) \rightarrow_{(G_P,G_T)} n'_s$，$ref_{G_C}(n_d) \rightarrow_{(G_P,G_T)} n'_d$，$label(n_a) = label(n'_a)$ とする．なお，(n'_s, n'_a, n'_d) が存在しない場合は何もしない
4. Redirection: redex の根を新規に追加された根に置き換える
   redex 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．

書換えでは明示的に頂点の削除を行う事は認めていない．ある書換えの結果では不要な頂点でも，別の書換え規則の観点からでは必要な頂点である可能性があるからである．そのため，どの有効な頂点からも参照されていない頂点を，不要な頂点と判断して削除するものとする．ここで有効な頂点とは，グラフに唯一存在する Root node から，接続を辿って到達可能な頂点とする．一度 Root node から到達できなくなった頂点は，どのような書換えを行っても Root node を根とする根付き連結グラフに接続される事はない．そのため，この削除方法は妥当と考えられる．

例 2.5 グラフ cons(car:0, cdr:append(car:cons(car:1, cdr:nil), cdr:nil)) に以下の書換え規則を適用する．

```
append(car:cons(car:E, cdr:L1), cdr:L2),
  listp(op:L1)
    -> cons(car:E, cdr:append(car:L1, cdr:L2))
listp(op:nil) -> true
listp(op:cons(cdr:C)) -> listp(op:C)
```

1. Matching: append(car:cons(car:1, cdr: nil), cdr:nil) が redex である．
2. Authentication: 述部には listp(op:L1) が存在する．そのため，書換え対象のグラフの複製に述部を構築する．

   ```
   (cons(car:0,
         cdr:append(car:cons(car:1, cdr:nil),
                    cdr:N[nil])),
    listp(op:N))
   ```

   述部の左辺 listp(op:nil) は true に書換え可能である．よって listp (op:N) は真であり，redex を書換えることができる
3. Building: 新たに次のグラフが構築される

   (cons(car:0,

```
        cdr:App[append](
            car:cons(car:One[1], cdr:N1[nil]),
            cdr:N2[nil])),
    Cons[cons](car:One,
                cdr:append(car:N1, cdr:N2)))
```

4. Redirection: 構築されたグラフにおいて，頂点 App に関して redex に現れない接続を全て Cons の接続として繋ぎかえる．

```
cons(car:0,
     cdr:Cons[cons](
          car:One[1],
          cdr:append(car:N1[nil],
                     cdr:N2[nil])))
```

このようにして得られたグラフが書換え結果である．App を根とするグラフは Root node から到達不能になるので回収された．

最後に接続内包表記の定義を与える．

定義 2.6 接続内包表記
規則右辺に含まれる接続内包表記

$$\mathrm{Node}_1(\mathrm{A}' : \mathrm{D}' \mid \mathrm{A} : \mathrm{D} \text{ in } \mathrm{Node}_2)$$

は Node_2 の持つ接続のうち，A:D にマッチするものに関して接続 (Node_1, A', D') を作成するものである．

2.4 階層グラフと分類

REGREL は階層グラフとよばれる REGREL グラフの階層を持つ．各階層は非負整数の Level を必ず 1 つだけ持っている．Level $i+1$ の階層に存在する書換え規則は Level i の階層に存在するグラフを書換える事ができる．特に Level 0 は Base Level とも呼ばれ，主に計算結果の表現を行う階層として用いられる．この階層構造によって，書換え規則と書換えられる対象を明確に分離することができる．このような構造を reflective tower [10] と呼ぶ．

Level i に属するグラフの項表現は図 2 の LGraph で表現される．ただし，level はそのグラフの Level を表す非負整数とする．省略した場合は Level 0 とする．この様に属する Level の情報が付加されたグラフを，Level 付きグラフと呼ぶ．

さらに，ある Level i に属するグラフの部分グラフに関して，その部分グラフに適用可能な書換え規則を限定できると便利な場合がある．そのような動作を表現するするための機構として分類を導入する．ある頂点 n を根とした根付き連結グラフが分類 $c_1,\ldots,c_j$ に属し，$c_{j+1},\ldots,c_k$ に属さないとき，$\{c_1,\ldots,c_j/c_{j+1},\ldots,c_k\}(n)$ の様に記述する．対象の根が明らかである場合には，頂点を囲む括弧を省略しても良い．それぞれの分類は固有のラベルによって名前付けされる．分類 c_1, c_2 が同じラベルを持つ場合，c_1, c_2 は同じ分類であるとする．ある Level $i+1$ に属し，分類 $c_1,\ldots,c_j$ に属する書換え規則は，Level i に属し，分類 $c_1,\ldots,c_j$ のどれかに属するグラフのみをその書換え対象とする．また，どの分類にも属さない書換え規則は，分類に属さないグラフのみを対象とする．

2.5 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. Import: Level 付きグラフを入力とし，そのグラフを適切な階層に追加する
2. Rewriting: 高位レベルから順に Base Level まで書換えを行う

   Level $n+1$ を書換えられなくなったら，Level n の書換えを行う．これを Base Level まで行う．
3. Print: Base Level のグラフを表示する．

Rewriting を行う際に、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．REGREL では安全に書換えを行うことができる redex 同士に関しては，同時並行に書換えを行うことを認めている．以下に安全な書換えの定義を示す．

定義 2.7 安全な書換え
書換え対象のグラフ G_T に含まれる複数の redex $R = \{G_{R_1},\ldots,G_{R_n}\}$ を，それぞれにマッチする書換え規則 $\{r_1,\ldots,r_n\}$ によって書換えるとする．このとき，R を安全に書換えることができるとは，どの $G_{R_i} \in R$ を r_i $(1 \le i \le n)$ で書換えても，以下の条件を満たす場合をいう．

- $R - \{G_{R_i}\}$ に含まれるどの redex の頂点，接続も削除されない
- $R - \{G_{R_i}\}$ に含まれるどの redex の根 n_r に関して，以下を満たす
  - どの接続 $(n_r, n_a, n_d) \in arcs(G_T)$ も削除されない
  - どの接続 $(n_s, n_a, n_r) \in arcs(G_T)$ も削除されない

書換えの際には，redex と書換え規則の組の集合から，安全に書換えを行う事ができる redex を非決定的に選択する．そしてそれらに対して，同時並行に書換えが行われる．

2.6 otherwise 規則

安全な書換えができない複数の redex に対して，どの書換え規則が適用されるのかは非決定的である．REGREL ではこの様な場合に，適用する規則を決定するための優先順位などは設けない方針である．しかし，実用上は優先順位がつけられると便利な場合がある．そのため，そのような機構の一つとして，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 otherwise 規則をオプションとして導入する．ここで，書換えが行えないかどうかの判断は分類毎に行われるものとする．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．また，otherwise 規則を認める場合の REGREL の動作は，以下の手順で行う．

1. otherwise 規則でない規則が適用できなくなるまで書換えを行う
2. otherwise 規則が適用できるならば，適用して (1) に戻る
3. 書換えを終了する

2.7 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る．ここでのアクターモデルの定義は [7] に準じる．

REGREL におけるアクターモデルではアクターとメッセージを頂点によって表す．アクターはその種類や状態を表すラベルを持つ．同様にメッセージもその種類を表すラベルを持つものとする．また，アクターやメッセージは既知のアクターへの接続を持つことが可能で，動作の際に利用することができる．このように，既知のアクターとの関係をグラフで表現した場合，DAG や循環を生じる場合があるが，REGREL グラフによって簡潔に表現することができる．

あるアクターを表す頂点 A にメッセージ M が送信されたことを，A(_*dst*:M(_*snd*:A)) で表すとする．また，アクター A がメッセージ M を受信したことを A(_*rcv*:M) で表すとする．なお，受信したメッセージを切り離すことを，A(_*rcv*!:M) で表す．そして，メッセージ送受信に関して表 1 の糖衣構文を認める．これらの糖衣構文に関して，(A ! M) のように括弧

表 1　メッセージ送受信に関する糖衣構文

項表現	糖衣構文
A(_*dst*:M(_*snd*:A))	A ! M
A(_*dst*:M1(_*snd*:A), ..., _*dst*:Mn(_*snd*:A))	A ! (M1, ..., Mn)
A(_*rcv*:M)	A ? M
A(_*rcv*:M1, ..., _*rcv*:Mn)	A ? (M1, ..., Mn)
A(_*rcv*!:M)	A / M
A(_*rcv*!:M1, ..., _*rcv*!:Mn)	A / (M1, ..., Mn)

表 2　グラフ操作クエリ

query	説明
lcollect(op:N, pattern:P)	頂点の線形探索
foreach(op:N, msg:M)	破壊的操作
reduce(op:N, ufunc:Fu, bfunc:Fb)	畳み込み
if(pattern:P, then:Mt, else:Me)	条件分岐
add	接続の追加
del	頂点の削除
cut(attr:_[A])	接続の削除
replace(node:N)	頂点の置き換え
rewrite(label:_[L])	頂点の書換え

で括った場合，アクターの根を表すものとする．例えば，あるアクターに対する送受信を同時に記述する場合には ((A ! M1) ? M2) / M3 のように記述することができる．

送信されたメッセージは書換え規則

```
A ! M -> A(_dst!:M(_snd!:A)) ? M
```

によって受信される．以下アクターモデルを利用する際には，暗黙にこの規則が用意されているものとする．

3　並行グラフ操作クエリ

本章では，REGREL の上で動作する汎用のグラフ操作クエリの定義を行う．このクエリは逐次動作が必要な部分ではそのように動作するが，基本的には並行に動作を行うものとして定義する．

そのクエリは大きく 3 種類に分類される．まず，選択された操作対象の収集や，それら対象に操作を行う基本クエリである．つぎに，条件分岐や逐次動作などを表す制御クエリ，そして頂点の追加や削除などを行うグラフ操作クエリがある．なお，主なクエリの一覧を表 2 に記した．

また，クエリは関数型とメッセージ型に分類される．関数型はクエリを根とする簡約を行い，その結果を計算結果とする．クエリから計算対象の頂点への op 接続を作成することで計算が開始される．メッセージ型はメッセージを受け取った頂点や接続に対して作用する．クエリによっては関数型，メッセージ型両方の使い方が可能なものもある．クエリを対象の頂点に送信する事で計算が開始される．

3.1　基本クエリ

3.1.1　lcollect クエリ

lcollect(op:N, pattern:P) は，頂点 N を根とするグラフがパターン P にマッチするとき，P の葉に対応する頂点全ての集合を返す関数型クエリである．ただし，パターン P に含まれる全ての頂点は，たかだか 1 つまでの接続しか持つことができない．集合は set 頂点を根とし，要素を elem 接続で保持するものとする．

```
lcollect(op:N, pattern:P) ->
  S[set](op:N ! mlcollect(pattern:P, root:S))
  N[L1] ? M[mlcollect](pattern:P[L2]),
    L1 !~ L2 -> N / M
  N[L] ? M[mlcollect](pattern:P[L], root:R),
    P !~ _(_:_) -> N / M(root:R(elem:N))
  N[L] ? M[mlcollect](pattern:_[L](A:P),
                      root:R),
    -> N(A:D ! mlcollect(pattern:P, root:R) |
         A:D in N) / M
```

lcollect は 1 つ目の規則によって set 頂点に置き換わると同時に，op 接続の終点全てに対して mlcollect メッセージを送信する．N ? M[mlcollect](patter:P) に関して，パターンの根 P と，メッセージを受信した頂点 N のラベルが一致しない場合，メッセージ M を切り離す．一方，一致する場合にパターン P が葉であるならば，頂点 N は探索対象であるので，根の set から頂点 N への elem 接続を作成して結果を保持する．また，パターン P を始点とする接続 (P, A, Ps) が存在するならば，N を始点とする全ての A 接続の終点 D に対して，メッセージ M を送信する．

3.1.2　foreach クエリ

foreach(op:N, msg:M) は頂点 N から到達可能な全頂点にメッセージ M を送信するメッセージ型と関数型の両対応クエリである．このクエリは対象の頂点を限定する機能は持たない．必要な場合は lcollect クエリであらかじめ対象を絞り込むか，送信するメッセージを制御クエリにする必要がある．

```
foreach(op:N, msg:M) -> N ! foreach(msg:M)
N ? F[foreach](msg:M),
  finished(lhs:F, rhs:N) !=> true
    -> (N(A:D ! F | A:D in N) ! scopy(op:M))
         / F(fin:N)
N ? F[foreach](msg:M),
  finished(lhs:F, rhs:N) -> N / F
finished(lhs:L(fin:N), rhs:R), N == R -> true
```

foreach メッセージは，処理を行った頂点を fin 接続によって保持している．N ? F[foreach](msg:M) に関して，(F, fin, N) が存在しない，つまり N が未処理の場合には以下の処理を行う．

- N にメッセージ F が持つメッセージ M を送信する
- N を始点とする全ての接続の終点 D に F を送信する
- (F, fin, N) を作成する

対して，(F, fin, N) が存在する場合，処理済みであるので単に N からメッセージ F を切り離すだけである．

3.1.3　reduce クエリ

reduce(op:N, ufunc:Fu, bfunc:Fb) は頂点 N から到達可能な全頂点に関数 Fu を適用し，得られた結果すべてを関数 Fb で縮約した結果の根を返す関数型クエリである．この処理では縮約される対象は非決定的に選択される．そのため，木の様に子の順序が重要になる場合には，他のクエリを利用する必要がある．また，関数 Fb は可換でなければならない．

```
R[reduce](op:N, ufunc:Fu),
  finished(lhs:R, rhs:N) !=> true
    -> R(op!:N, fin:N, op:D | A:D in N,
         elem:scopy(op:Fu(op:N)))
R[reduce](elem:E1, elem:E2, bfunc:Fb)
  -> R(elem!:E1, elem!:E2,
       elem:scopy(op:Fb(lhs:E1, rhs:E2)))
R[reduce](elem:E),
  R !~ _(op:_), R !~ _(elem:_, elem:_) -> E
```

reduce 頂点は fin 接続によって，関数 Fu を適用した頂点を保持する．R[reduce](op:N, ufunc:Fu) に関して，N が処理済みでなければ以下の処理を行う．

- (R, op, N) を切断する
- (R, fin, N) を作成する
- R から scopy(op:Fu(op:N)) への elem 接続を作成する
- N の持つ全ての接続の終点 D に対して (R, op, D) を作成する

一方，(R, fin, N) が存在する場合，N は処理済みであるので単に (R, op, N) を削除する．関数 Fu を適用した結果が 2 つ以上存在する場合，それらを関数 Fb で縮約して 1 つにまとめる．全ての頂点に関数 Fu が適用され，その結果を関数 Fb で 1 つに縮約された場合，それを結果として返す．

3.2 制御クエリ

主な制御クエリとして，ここでは if クエリの説明を行う．書換え規則は付録 A に記した．

if(patter:P, then:Mt, else:Me) は条件分岐を行うためのメッセージ型クエリである．N ? I[if](pattern:P) に関して次のように動作する．

- パターン P が N にマッチし，(I, then, Mt) が存在する場合，N に Mt を送信する
- パターン P が N にマッチせず，(I, else, Me) が存在する場合，N に Me を送信する
- それ以外の場合は何もしない

その後，N から I を切断する．前節で述べたように，基本クエリは対象を選択する能力を持たないので，制御クエリを持ちいて対象を限定する必要がある．

3.3 グラフ操作クエリ

主なグラフ操作クエリとして, add, cut, del, replace，そして rewrite の説明を行う．書換え規則は付録 A に記した．

add はメッセージ型のクエリである．頂点 N がメッセージ M[add] を受信したとき，M がもつ全ての接続 (M, A, D) に関して，接続 (N, A, D) を追加するように動作する．cut(attr:_[A]) は，それを受信した頂点 N に関して，属性 A を持つ接続を全て削除するメッセージ型のクエリである．

del は，これを受信した頂点を削除するメッセージ型クエリである．実際には直接頂点を削除することはできないので，このメッセージを受信した頂点への接続を全て削除していく．replace(node:N) は，受信した頂点を根付き連結グラフの根 N で置き換えるメッセージ型クエリである．rewrite(label:N[L]) は，受信した頂点のラベルを頂点 N の持つラベル L に書換えるメッセージ型クエリである．REGREL は頂点のラベルを直接書換える事はできないので，実際は replace(node:_[L]) と等価である．

3.4 計算順序制御

クエリを正しく処理するには，任意の順序でクエリを簡約してはならない．例えばクエリ `reduce(op:foreach(op:...))` では，reduce よりも foreach が先に簡約されなければならない．つまり，最内簡約される必要がある．

このとき，最内簡約で簡約が進むようにするには，クエリ毎の規則をそのように動作するように書換える必要がある．その場合，書換え規則は書換えによる計算の意味と，評価戦略の両方の情報を持つことになる．しかし，クエリを定義する書換え規則はその意味のみを定義し，手動で規則の修正を行わずに評価戦略を与えられると便利である．以下に，分類と高階書換えを用いて最内簡約が行われるように変更する枠組みを示す．

まずクエリの計算に用いる書換え規則を元に，redex の根を発見する書換え規則と，発見された redex を 1 度だけ書換える書換え規則を作成する．この操作は，対象の書換え規則を分類 rules に入れることで，以下の高階書換え規則で行われる

```
{rules}((L -> R) ->
  {root/rules}(pair(
    fst:{reduction}(L ? Re[redex] -> R / Re),
    snd:{marking}(L, L !~ (_ ? redex)
                  -> L ! redex))))
```

分類 rules に入れられた書換え規則を元にして，redex メッセージを受信した時のみ，1 度書換えが行われる書換え規則を分類 reduction に作成する．同時に，元の書換え規則の規則左辺 L を持ち，L の根に redex メッセージを送信する書換え規則を分類 marking に作成する．特に，後者の書換え規則をマーカと呼ぶことにする．両者の規則は，pair 頂点を根とする根付き連結グラフの部分グラフとして表現されている．本来は元となる規則から rules, marking 分類にそれぞれの書換え規則を生成するべきであるが，書換えの結果として複数の根付き連結グラフを生成することが認められていないため，この様な表現を行っている．

前準備が終了したら次の規則で最内簡約を行う．

1. 計算対象の選択: 計算対象の根に eval メッセージを送信する
2. 計算開始: eval を受信した頂点を分類 marking に入れる．
3. redex の探索: マーカを用いて書換え対象候補の根に redex メッセージを送信する．
4. 最内 redex の探索: redex メッセージを受信した頂点への接続，または outer メッセージを受信した頂点への接続をもつ頂点に outer メッセージを送信する．この時，redex を受信したが outer を受信していない頂点が最内クエリである．
5. 終了判定: eval メッセージを受信した頂点が，outer メッセージを受信していない場合，redex が存在しないため計算は終了している．
6. redex の簡約: redex メッセージと outer メッセージを受信した頂点から redex メッセージを切り離す．次に，分類 marking の書換えが行えなくなったら eval メッセージを受信した頂点を分類 reduction に入れ，分類 marking から外す．そうすることで計算対象のグラフ全体が分類 reduction に入る．これによって，分類 reduction のグラフで，redex メッセージを受信している頂点を根とする部分グラフそれぞれが，前準備によって作成された分類 reduction の書換え規則によって 1 度書換えられる．
7. 次の計算の準備: outer メッセージを受信した頂点から，そのメッセージを切り離す．書換えられなくなったら，eval メッセージを受信した頂点を分類 marking に入れ，分類 reduction から外す．そうすることで計算対象のグラフ全体が分類 marking に入る．これによって，全体から redex を探索する処理に移る．そして 3 に処理を戻す．

この動作を行うための書換え規則を付録 B に記す．

3.5 実例

本節ではクエリを用いた実例を示す．

前準備として，特別な頂点と書換え規則として，浮動小数点数を定義する．浮動小数点数は，正規表現-?[0-9]+(

.[0-9]+)?で表されるラベルを持つ頂点で表すものとする．+, -, mul, div, pow はそれぞれ接続 lhs, rhs を持つ 2 項演算子で，2 数の和，差，積，商，累乗を表す頂点に書換えられる．eq, neq, lt, le, gt, ge はそれぞれ接続 lhs, rhs を持つ 2 項演算子で，比較演算 $=, \neq, <, \leq, >, \geq$ の結果 true, false に書換えられる．また nump は接続 op を持つ単項演算子で，引数が浮動小数点数ならば true，そうでなければ false に書換えられる．

次に，グラフの複製を行う演算子として scopy, dcopy を定義する．それぞれ接続 op を持つ単項演算子で，引数の浅い複製，深い複製に書換えられるとする．

例 3.1 要素の探索
ここでは XML[11] における XPath[12] の例をいくつか挙げ，REGREL で同様の操作を行うクエリを示す．XML は接続にラベルを持たない構造であるので，REGREL の例では属性によらず探索を行うものとする．

- 根から foo, bar, baz 要素と続く場合の baz 要素

```
XPath: /foo/bar/baz
REGREL: lcollect(pattern:
                 foo(_:bar(_:baz)))
```

- 根以下に存在する全ての要素

```
XPath: //*
REGREL: reduce(ufunc:mkset, bfunc:union)
        mkset(op:N) -> set(elem:N)
        union(lhs:S1, rhs:S2)
          -> S1(elem:E | elem:E in S2)
```

- 根以下に存在する全ての要素数の和

```
XPath: (//node()).length
REGREL: reduce(ufunc:one, bfunc:+)
        one(op:_) -> 1
```

- 根以下で，親に bar 要素を持つ baz 要素

```
XPath: //bar/baz
REGREL: reduce(ufunc:lcollect(
                  pattern:bar(_:baz)),
               bfunc:union)
```

例 3.2 平均最短距離の計算
根付き連結グラフの平均最短距離を求める．計算は次の流れで行う

1. 対象のグラフの全頂点数を計数する

```
reduce(ufunc:one, bfunc:+)
```

2. 根から到達可能な全頂点 (根を含む) を集めた集合を作る

```
reduce(ufunc:mkset, bfunc:union)
```

3. 2. で得られた集合に対して，それぞれの頂点を根とした根付き連結グラフの深い複製を作成し，その頂点が持つラベルを全て undef に書換える

```
map(msg:if(pattern:set,
           else:rewrite(label:undef)))
map(op:N) -> foreach(op:dcopy(op:N))
```

4. 3. の集合に対して，それぞれの根からの最短距離を求め，その距離でグラフのラベルを置き換える

```
foreach(msg:if(pattern:set, else:spl))
S[spl], S ~ _(length:_) -> S(length:0)
U[undef] ? S[spl](length:L)
  -> L(A:D ! spl(length:+(lhs:L, rhs:1)) |
       A:D in U) / S
N ? S[spl](length:L), gt(lhs:N, rhs:L)
  -> L(A:D ! spl(length:+(lhs:L, rhs:1)) |
       A:D in N) / S
N ? S[spl](length:L), le(lhs:N, rhs:L)
    -> N / S
N ? S[spl], nump(op:N) !=> true -> N / S
```

5. 3. の集合の各要素に対して最短距離の総和を求める

```
reduce(ufunc:num, bfunc:+)
num(op:set) -> 0
num(op:N), N !~ set -> N
```

6. 1. と 4. から最短距離の総和を (全頂点数 - 1) の二乗で除した結果が平均最短距離である．

この動作をまとめると以下のクエリが構成される．ただし対象とするグラフの根を N で表す．

```
div(lhs:reduce(ufunc:num, bfunc:+, op:
      map(msg:if(pattern:set, else:sp),
          op:reduce(ufunc:mkset, bfunc:union,
                    op:N))),
    rhs:pow(lhs:-(lhs:reduce(ufunc:one,
                             bfunc:+, op:N),
                  rhs:1),
            rhs:2))
```

4 実装

2 章で示した仕様をもとに実装を行った．実装は Python 2.6 を用いて行った．処理系のソースとサンプルプログラムを http://bitbucket.org/htk16/regrel から入手可能である．この処理系は，レベル付きグラフの項表現を入力として受け取り (Import)，書換え (Rewriting) を行った後 Base Level のグラフを表示する (Print) を繰り返すインタプリタとして実装されている．

5 考察

5.1 グラフ書換え系の比較

ここでは，REGREL と，既存のグラフ書換え系である DACTL [1]，LMNtal [8, 9] との比較を行う．p 基本的には，DACTL, LMNtal 共に REGREL と同じく，適用可能な書換え規則が非決定に選択され適用される．だが，実際の計算を表現する場合には，適用される書換え規則の限定や，適用される順序の制御は必須と言っても良い．以下，主に書換え規則の適用を制御する機能に着目して比較を行う．

DACTL は頂点にラベルを持つ無向グラフの書換え系である．REGREL と大きく異なる点は，頂点の種類毎に出次数が固定であることである．また，パターンマッチでは部分マッチを行うことができない．そのため，規則左辺に現れる頂点は，持っている全ての接続を記述する必要がある．この点では REGREL の規則の方が記述力が高いといえる．

評価戦略に関して書換え規則の特殊化を行う為に，DACTL では Control Marking と呼ばれる組み込みの制御機構を導入している．この Control Marking を用いることで，書換えの根となる事ができる頂点を明示する事ができる．だがこの方法では，書換え規則毎に，次にはどの頂点が根となるのかを明示しなければならない．しかも，Control Marking を用いる書換え規則は，任意の順序で書換えを行う書換え規則とは全く別に定義する必要がある．一方 REGREL では 3.4 節で示したように，分類を用いた書換え規則を用いることで，最内簡約を行う枠組みを定義することができた．その枠組みは，

ある分類に属するグラフを簡約し，停止次第別の分類に移動させるという方法で実現されている．クエリの簡約を行う規則は，特定の分類に所属させるだけで良く，別途定義をしなおす必要は無い．この様に，意味を表す書換え規則と評価戦略を決定する書換え規則を直交した形で表現することができる．そのことから書換え規則を適用する順序制御の観点では，REGREL の方が表現力が高いと考えられる．

LMNtal は頂点と接続にラベルを持つ無向グラフの書換え系である．DACTL と同様に頂点の種類毎に出次数が固定で部分マッチを行う事はできない．そのため，書換え規則の規則左辺の記述力に関しては，REGREL に利点があると考えられる．

LMNtal では膜と呼ばれる集合を扱う事ができるのが特徴である．膜はグラフ，書換え規則そして膜自体を要素として取る集合である．また，膜に含まれる書換え規則はその膜内でのみ有効である．よって膜を用いることで，書換え規則の適用範囲を限定することが可能である．そして，膜は入れ子構造を取ることができるので，ある膜に含まれる部分グラフにのみ適用可能な規則を定義する事ができる．この動作に関しては，REGREL においても対象とするグラフの一部のみを計算対象の分類に入れることで表現可能である．ただし，LMNtal は任意の深さの膜入れ子を作成することが可能であるのに対して REGREL では分類の種類を増やすことは簡単ではない．なぜならば，書換えられるレベルと，規則の属するレベルでそれぞれ分類を作成する必要があるためである．

LMNtal では膜内の要素を別の膜に移す書換え規則は，移動元と移動先両方の膜を対象にできる位置に存在しなければならない．その上，その両者の膜をパターンで記述する必要があり，規則右辺では膜間の要素移動を記述する必要がある．REGREL はパターンには移動元の分類だけが必要で，規則右辺では移動元と移動先の分類を記述するだけでよい．そのため，この様な要素の所属先を変更する書換えは REGREL の方が記述しやすいと言える．また LMNtal では包含関係が無い膜に同一の規則を所属させるには，それぞれ独立したグラフを生成する必要がある．一方，REGREL では，あるグラフが複数の分類に横断して属する事ができる．

以上から分類を用いた手法と，膜を用いた手法は一長一短であるといえるだろう．LMNtal の膜の様に，手軽に新たな分類を作成し利用する機構があるとより表現力を増すことができるであろう．

5.2 汎用グラフ操作クエリの比較

XQuery[5] や LINQ[6] のような木構造やグラフの汎用操作クエリは基本的に逐次的に動作するように設計されている．一方，今回提案した REGREL 上で動く操作クエリ言語は，基本的に並行動作する．また逐次処理が必要な場合は，最内部分の redex のみが書換えられるように動作する．そのため，並行動作を並行リソースが有効に利用できると考えられる．しかし，クエリの記述性には難がある．例えば lcollect のように，あるパターンに従った頂点の収集を行うクエリでは，XPath の方が記述量が少なく簡潔である．そのため，実用的にはクエリの記述は XPath 等を用いて行い，計算時にはその内容を REGREL 上で定義した汎用グラフ操作クエリに置き換えてから実行できれば，記述性の高さを保ったまま並行計算が可能な箇所の割合を上げる事ができると考えられる．

6 おわりに

提案するグラフ書換え言語 REGREL の定義を示した．そして，実例としてグラフの操作クエリ言語を挙げ，分類と呼ばれる書換え規則の適用範囲を限定する機構と，高階書換えを用いることで，手作業で修正することなく書換え規則の動作を評価戦略に関して特殊化することができた．また，定義したクエリは基本的に並行に動作し，明示せずとも必要な箇所では逐次処理や同期を行うように動作させることができた．

参考文献

[1] John R. W. Glauert, Richard Kennaway, George A. Papadopoulos, and Ronan Sleep. "dactl: An experimental graph rewriting language". In *Proc. 4th International Workshop on Graph Grammars*, pp. 378–395. Springer-Verlag, (1991).

[2] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *"Term Rewriting Systems"*. Cambridge university press, (2003).

[3] M.R.Sleep, M.J.Plasmeijer, and M.C.J.D. van Eekelen. *"Term Graph Rewriting"*. Wiley Professional Computing, (1993).

[4] 井田哲雄. "計算モデルの基礎理論". 岩波書店, (1991).

[5] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon, editors. *"XQuery 1.0: An XML Query Language"*. W3C, (2007).

[6] Don Box and Anders Hejlsberg, editors. *"LINQ: .NET Language-Integrated Query"*. MSDN, (2007).

[7] 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).

[8] 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).

[9] 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).

[10] Pattie Maes and Daniele Nardi, editors. *"Meta-level Architectures and reflection"*. North-holland, (1988).

[11] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau, and John Cowan, editors. *"Extensible Markup Language (XML) 1.1 (Second Edition)"*. W3C, (2006).

[12] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon, editors. *"XML Path Language (XPath) 2.0"*. W3C, (2007).

付録 A クエリを表現するの書換え規則

```
N ? I[if](pattern:P, then:Mt),
  N ~ P -> (N ! Mt) / I
N ? I[if](pattern:P),
  N ~ P, I !~ if(then:_) -> N / I
N ? I[if](pattern:P, else:Me),
  N !~ P -> (N ! Me) / I
N ? I[if](pattern:P),
  N !~ P, I !~ if(else:_) -> N / I

N ? Add[add] -> N(A:D | A:D in Add) / Add
P(A:N ? del) -> P(A!:N)
N ? C[cut](attr:_[P])
  -> N(A!:D | A[P]:D in N) / C
_ ? replace(node:N) -> N
_ ? rewrite(label:_[L]) -> [L]
```

付録 B 最内簡約を行うための書換え規則

```
N ? eval -> {marking}N
{marking}(N(_:_ ? redex), N !~ (_ ? outer)
            -> N ! outer)
{marking}(N(_:_ ? outer), N !~ (_ ? outer)
            -> N ! outer)
{marking}(N ? (Re[redex], outer) -> N / Re)
{marking}(R ? (eval, outer)
            |-> {reduction/marking}R)
{reduction}(N ? O[outer] -> N /O)
{reduction}(R ? eval
              |-> {marking/reduction}R)
```