プログラミング 13−1
(1997. 5. 21)

# イベント駆動型グラフ書き換え言語

原田康徳
NTT 基礎研究所

イベントによって制御されるグラフ書き換え型言語を提案する．一般に，項書き換え，グラフ書き換えシステムでは，依存関係のない書き換えを許しているが，この言語では依存関係に従った書き換えに制限することで，高速な実装を実現している．最初に，プログラミング可能な図形エディタを記述するために，イベント駆動型図形書き換え言語 VISPATCH を作成した．その経験を元に，グラフの表示上の問題を避けるために，表示の機能を取り去った，純粋なグラフ書き換え型言語を作成した．これらの言語の大きな特徴は，名前を用いないという点である．一般にプログラミング言語の構文は BNF で定義されるが，これは木構造しか表現できない．しかし，プログラミング言語本来がもっている構造はグラフでなければ表現できないので，それを補うために名前が用いられた．同じ名前のついた葉は同じ葉を参照しているという解釈を与えるのである．グラフ言語を考えることにより，この余計な解釈から解放されて，プログラミング言語が本来持っていた構造をじかに考えることができる．

# Event Driven Graph Rewriting Languages

Yasunori Harada
NTT Basic Research Laboratories

We propose two event driven graph rewriting languages. An event is used for restricting rewrite-location and timing. One is a rule-based visual programming language (VISPATCH). And the other is a pure graph based-programming language that has no visual representation problem. They are programming languages without names, because of a program structure is directly represented by a graph structure.

## 1 研究の背景

イベントで書き換えが制御されるグラフ書き換え型言語を2つ提案する．最初に我々は，図形エディタなどのGUIアプリケーションを記述するために，ユーザイベントに反応するビジュアル言語VISPATCHを開発した[1]．それは，ユーザのイベントが生じたときに，そのイベントの周辺の図形にマッチする書き換えルールが探され，それによって書き換えが行われるような言語である．対象はベクトルグラフィックであり，図形をエッジとしたグラフ構造をしている．イベントは書き換えのタイミングと場所を制限する働きを持っている．書き換えの結果，別のイベントを発生させることができ，それは新たな書き換えを引き起こす．イベント以外で書き換えは起動しないので，依存関係によった書き換えのみを行なういうことである．

依存関係のある書き換えとは，prologでの数式微分のプログラムを考えればよい．ある項を微分するときは，まず部分項を微分して，それらを用いて自分を微分した項を作る．部分項を微分するために，新たに書き換えが必要であるが，これは依存関係に従った書き換えである．一般に項書き換え，グラフ書き換えシステムと呼ばれているものは，これより広く強力な，依存関係に無関係な書き換えを行なうことができるシステムである．これらで，依存関係に従って書き換えを行なうためには，全ての書き換えルールのヘッドに特別な項(エッジ)を含ませて，その項(エッジ)の有無で書き換えを制御すればよい．しかし，この強力さが効率のよい処理系の構築の妨げになっている．

VISPATCHは，グラフを対象にしていることとで，パターンマッチングやデータ構造は項を対象にするよりも複雑になるのであるが，一方で依存関係のある書き換えに制限しているので，処理系の実装は容易である．

VISPATCHのもう一つの特徴は，書き換えルール自身も図形で表現し，それを書き換えの対象にできるようにしたことである．これにより，自己拡張可能なシステムが実現できる．

VISPATCHは名前を使わないプログラミング言語である．名前とは，通常のプログラミング言語における，関数名や変数名のことである．テキスト(もしくはBNFによる文法の定義)は直接には木構造しか表現できない．しかし，実際のプログラミング言語が表現したい対象はグラフ構造が必要であるので，離れた場所を関係つけるために，名前が導入されたのであった．しかし，VISPATCHでは離れた2点を図形(例えば直線)により直接関係つけることができるため，名前を使用する必要が無くなったのである．

プログラミング言語において名前を使用しない，ということは大きな利点である．例えばラムダ式の変形では変数名の衝突に気をつけて行わなければならない．同じ名前が表れたら，どこにも使われていない新しい名前に置き換える必要がある．これはラムダ式が木で表現され，同一の変数が名前を用いて間接的に指示されていたからである．最初からラムダ式をグラフで表現しておれば(つまり同一の変数は同じ点を共有する)，ラムダ式の変形などは単なるグラフの置き換えで済んでしまう．同様に名前にまつわるプログラミング言語上の問題(例えば，変数のスコープ，モジュールプログラミングなど)もグラフ表現では表れてこない．このように，名前を使用しないプログラミング言語では，プログラムの本質を直接みることができる．この性質の積極的な一つの応用はプログラムを操作するプログラム，例えばコンパイラの記述である．とくに，部分計算や最適化などの処理には高い能力を発揮すると考えられる．

こういった大規模な応用に耐える為には，図形言語はよいアプローチではない．図形言語でのプログラミングでは，2次元グラフの配置という，言語のモデルからは本質的ではない問題がある．プログラミング言語のモデルとユーザインタフェースとを分離して，独立に解決するべきである．後者はプログラムをいかに人間に読みやすく，書きやすくするかという問題を解決し，前者の問題をよりシンプルにすることができる．

こうして，図形言語から表示部分の機能を取り去り，図形の表示からの様々な制約から解放されたシンプルなグラフ言語が誕生した．

次の節では，最初に開発された図形言語VISPATCHを簡単に紹介する(詳しくはホームページ参照)．次に表示の問題を無視したグラフ言語を提案し，最後にそれらの応用を述べる．

## 2 VISPATCH(Visible Event Dispatch)

VISPATCH の基本構造は次のようである.

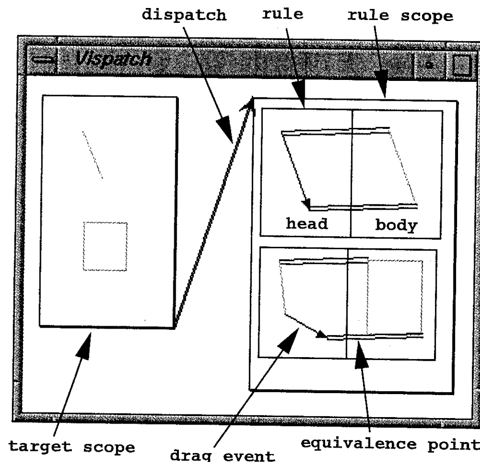


図 1: Simple program of VISPATCH

左側の矩形はイベントをセンスする領域(対象領域)で，そこで生じたマウスイベントはその右下から出ている太い矢印記号(Dispatch 図形)を伝わり，その先のルール領域(右側の矩形)で解釈，実行される．ルール領域には複数のルールを記述することができる．ルールの左側(ヘッド)にはイベントの周辺とマッチさせる図形が書かれており，マッチに成功すると，ルールの右側(ボディ)の図形と置き換える．ヘッドとボディとで同一の座標を表現するために，同一点図形を用いて，両者の点を結ぶ．ヘッドには一つのイベント図形が含まれており，それがマッチングに使用される．一方，ボディにイベント図形が含まれる場合にはそれを描画する代わりに，イベントを生成し新たな書き換えを引き起こす．これは再帰図形の生成や繰り返し操作の定義に使える．

図 1ではルール領域の中で 2 つのルールが定義されている．上のルールは何もないところでドラグイベントが生じると，それに沿って直線を生成する．下のルールは直線の端点でドラグイベントが生じると，その直線を削除して，矩形を生成する．

ボディで生成するイベントを，別のルールセットを用いて書き換えを行いたい場合に，イベント図形からそのルールセットへ Dispatch 図形を用いて指示する方法が用意されている．ある対象領域でのマウスの動作を様々に切り替える為に(図形エディタでのツールの持ち換えに相当する)，対象領域から出ている Dispatch 図形を様々なルール領域に向け直す仕組みが必要であるが，それ自身も書き換え行うことができる．

イベント図形，同一点図形，Dispatch 図形など，ルールの中で特別な意味を持つ図形を，書き換えの対象にするために，パターン中のそれらの特別な働きを禁止する仕組みが必要である．VISPATCH では図形の色の情報を利用して，それを行っている．ルール図形の中に含まれる図形の中でルール図形と同じ色を持つものは特別な働きをするものとして，マッチングや生成の対象としては使用しない．2 つの色を用意して，それぞれの色のルールがあればお互いのルールを編集することが可能となる．

そのような機能を利用して作成されたのが，図 2である．これは，自分自身で拡張が可能とな図形エディタである．図形エディタ全体を囲む矩形が一つの対象領域であり，そこからの Dispatch 図形は一度エディタの中心と通って，その周辺のどれかのルール領域に向けられている．その方向を向け直すためのルールが右下にあり，その対象領域が編集用のルールの周りの 4 つの矩形となっている．そこにある直線の端点をクリックすることで，その直線の先に中心からでている Dispatch 図形が向けなおされる．それによってツールの持ち換えが実現できた．左右にある縦に並んでいるルール群はプリミティブの図形を生成するルールで，それぞれで 2 色のすべての種類の図形を生成する．上に並んでいる 2 つのルールは図形の消滅，変形を行うもので，任意の図形にマッチするワイルドカードパターンが用いられている．以上が，ブートストラップとして最初に用意されていなければならない図形である．これらを用いて自分自身で新たにルールを拡張することができる．例として，下に書かれている 2 つのルールは，2 色のルールのテンプレートを描画するルールで，このエディタ自身で作成された．

## 3 図形によらないグラフ言語

VISPATCH では，新しく点を生成するときに，その座標をどのように決定するかを指定しなければ

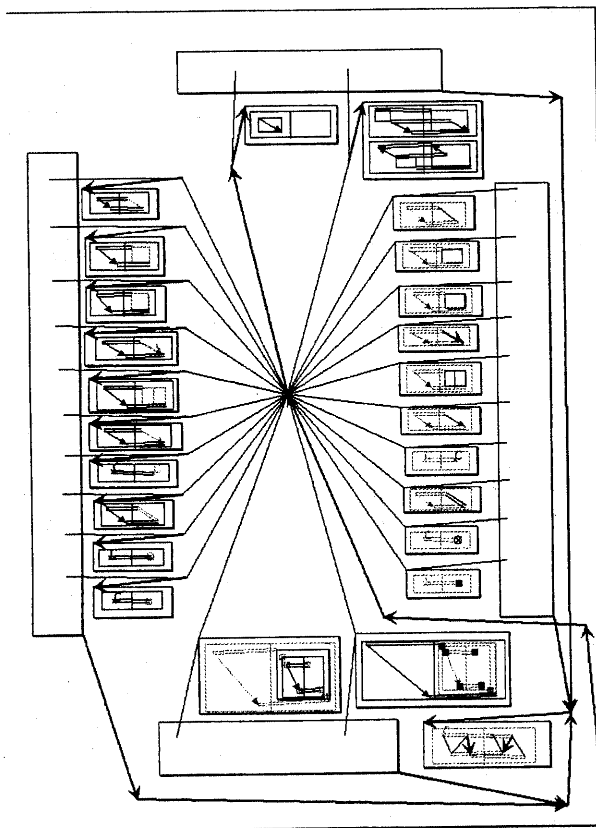

図 2: Fully programmable graphic editor.

ならない．例えば，ルールボディと相似な図形を生成するなどである．それは図形エディタを計算の対象している場合には本質的なことであった．しかし，一般的なグラフのマッチングによる計算という問題に VISPATCH を使用するために，その座標という性質が煩わしい．

また，図2でもわかるように，プログラム全体の構成はそれほど難しくないが，互いの図形が交差しないような配置を考えたりするのが面倒である．例えばルールを編集するときと，それらを使用するときとでは表示の仕方を変えるような仕組みが欲しい．しかし，言語の本質からは離れた問題である．

そのような理由から，点の座標，エッジの表示の能力を取り去ったグラフ言語を考えることにする．表示に関しては別のシステムにまかせることにして，ここではふれない．

まず，グラフのエッジを「1 つ以上の点からなり (ハイパーグラフ)，その順序に意味を持たせる (有向)．」と定義する．VISPATCH では図形の種類や色を操作する機能は提供されていなかった．それで，図 2 のようにあらかじめすべてのプリミティブ図形を 2 色分生成するルールを用意する必要があった．また，ワイルドカードによるマッチングにあったような拡張も考えられるが，きりがないし，アドホックでもある．それに対して，エッジが 1 つ以上の点を含むことができるようにしたことで，簡単にすべての情報をエッジのつながり方だけで表現することができるようになった．例えば，エッジの種類は，

特定の点を共有していることにする．それで3つの点からなるエッジの第1の点をそのエッジの種類を表すことに使用し，残りの2点で開始点と終了点を表現すれば，エッジに名前がついた有向グラフとなる．これらの関係はすべて書き換えで変更可能なので，エッジの種類や色を書き換えで変更できることに相当する．

グラフ言語は，名前を一切使わない表現形式である．しかし，テキストで表現する方法があると扱いに便利であるので，テキストでの表記を考えることにする．ここでは，点に名前をつけてそれをエッジが参照する形でグラフを定義する．サブグラフによる抽象化を用意して，点の名前はそのサブグラフに閉じたスコープを持つ．この欠点は，サブグラフの抽象化ができないくらい複雑に絡まったグラフを表現することができないことである．しかし，それまでの間この表記は使える．注意しておくが，これは単なる一つの表現形式にすぎないということである．

```
ex1(root x y z * +) {
  (* root a z)
  (+ a x y)
}
ex1'(root x y z * +) {
  (* root (+ # x y) z)
}
ex1''(root x y z * +) {
  (+ (* root # z) x y)
}
```

ここで，( ) { } _ #及び空白/改行は特殊な意味をもつ記号である．それ以外のすべての文字は名前を構成するために自由に使用できる．名前はサブグラフと一つのサブグラフ内で閉じたノードを参照するのに使われる．ex1(..) {...}は一つのサブグラフを定義している．サブグラフは外からいくつかの点を渡され，それをex1(..)内の名前(仮引数)で受ける．{...}はサブグラフの本体で(..)は一つのエッジを表している．ex1は，外部から6つの点をもらい，それを用いて2つのエッジを定義している(図3)．仮引数に存在しない名前(ここではa)は，サブグラフ内部だけで参照されている点である．特別に_という名前は無名の点として，複数個表れても互いに共有しない．

次のex1' とex1'' はいずれもex1と同じサブグラフを別の方法で表現している．ex1ではaという名前の点が外部から隠蔽され，2つのエッジ(これをe1, e2とする)で共有されていた．この様な隠蔽されて2点でのみ共有されている点があるとき，その名前を省略する記法を用意する．一方のエッジe2を他方のエッジe1のaのところに埋め込み，e2のaの場所に#を書く(ex1')．e1とe2の関係は対等であるので，反対に埋め込む方法もある(ex1'')．ここで，ex1'はS式の表記(* (+ x y) z)に似ていることに気付くであろう．違いはエッジの第2番目が自分の親を指しているということである．一方で，ex1''の表記はデータフロー的な読み方ができる．つまり，最初にxとyに対して+の計算を行い，それを受けてzとの*の計算を行ない，その答をrootに返している．いずれも表記の違いであるだけなのに，人間にとってはずいぶんと違った印象が与えられる．これが，グラフ言語を考える一つの狙いである．グラフ言語によってプログラミング言語間の比較が容易になる．

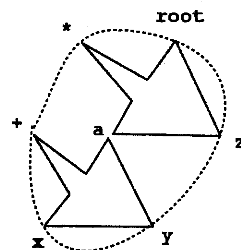


図 3: Graph Representation

## 4 イベント駆動型グラフ書き換え言語

図形によらないイベント駆動型グラフ書き換え言語を示す．まず最初にルールの構文を定義する．一つのルールは5点のエッジで表現する．

```
(prev head body event next)
```

最初の点はこのルールを参照し，続く2つの点はヘッドとボディのパターンを参照するためにつかう．次にボディでのイベント生成を示す特殊なエッジを示

す点，そしてこの後に続くルールを参照する点で構成されている．head と body は同じ長さのエッジを参照し，それらの対応する点がヘッドとボディのそれぞれのパターンで同一な点であるとみなされる．例えば，$0+a \rightarrow a$ という式の変形のルールは，

```
rule1(prev next) {
 (prev (# +  0  s  (+ # 0 a) a )
       (# +' 0' s' a'         a')
       _ next) }
```

のように表現できる．ヘッドパターンには一つのエッジ(+ # 0 a) がある．書き換えによって，それを消去してa' と置き換える．ヘッドとボディのパターンはルールエッジ以外とはどことも接続していない点に注意(そのために，ヘッドとボディとでノード名+, 0 などが違う名前になっている)．それによって，ヘッド，ボディの各パターンの範囲が明確になる(VISPATCH ではその範囲をルール図形の領域で示していた)．

次に，書き換え $s(a)+b \rightarrow a+s(b)$ と次の書き換えイベントを発生させるルールは

```
rule2(prev next) {
 (prev (# +  0  s  head a  b )
       (# +' 0' s' body a' b')
       event next)
 (+  head (s # a) b)
 (+' body a' (s' # b'))
 (event (# +' 0' s' body) prev)
}
```

となる．ここで，ヘッドには(+ head (s # a) b)，ボディには(+' body a' (s' # b')) というエッジがある．また，マッチすると新たにイベントを(event (# +' 0' s' body) prev) によって生成する．

イベントは一つのエッジ(参照点エッジ)と，ルール列との組である．書き換えはまず，参照点エッジの2番目以降の点と，ルールのヘッドエッジとの対応をとる．ヘッドエッジの方が長くて，余った点はヘッドルール中の自由な点で(ここでは a と b)，それに対応する点をボディで参照できる(a' と b')．ヘッドパターンに含まれるエッジが矛盾無くすべて対象に存在すると，マッチングに成功し，それらのエッジを取り除き，対応するボディパターンのエッジを生成する．さらにルールエッジの4番目の点(event)を先頭に持つようなエッジが存在すると，それをイベント生成命令として扱う．

上で定義したルールを起動するイベントは，参照点エッジ(記号を示す点(+, 0, s)と，書き換えたい式を参照する点，からなる)と，書き換えに使うルール列で構成される．これらを呼び出すには，

```
(event (# + 0 s (+ (s # (s # (s # 0)))
                   (s # 0)))
       rule1(# rule2(# _)))
```

となる．ここで，rule1( .. ) のように，サブグラフ名の直後に(があるときにはそのサブグラフのインスタンスを生成する．ここでも# の略記法を使うことが出来る．

## 5 考察とまとめ

図形書き換え型言語として ChemTrains[2] があるが，ユーザのインタラクションを考慮していない点と，ルール自身を書き換えの対象にできない点で，異なっている．

2つのイベント駆動型グラフ書き換え言語を提案した．VISPATCH は WWW ブラウザを用いて http://www.brl.ntt.co.jp/people/hara で動作を確認できる．

## 謝辞

VISPATCH の共同研究者である，NTT 基礎研究所の宮本健司氏，尾内理紀夫リーダーに感謝いたします．

## 参考文献

[1] 原田，宮本: *Visible Dispatch: visibility* に基づくアプリケーション構築法，WISS'96.

[2] Bell, B. and Lewis, C. : *ChemTrains: A Languages for Creating Behaving Pictures*, IEEE Symposium on Visual Languages, 1993.