# 再帰的なグラフパターンに基づく反復パターンマッチングの効率化手法

白井 涼也　今川 連　山本 直輝　上田 和紀

プログラミングやモデリングの対象には，代数的データ構造を超えたグラフ構造の表現や，グラフ構造に対するパターンマッチング機能と変換機能が有用となるものも少なくない．このような機能を備えたグラフ書換え言語のうち，CSLMNtal は，強力なパターンマッチング機能を提供する点を特徴としており，組み込みの型に加え，ユーザが再帰的に定義した型を用いてパターンマッチを行うことでグラフを書き換えることができる．素朴な実装では，パターンマッチに成功しグラフを書き換えるたびに，書換え後のグラフについて最初からパターンマッチを行っていたため実行時間が大幅にかかってしまう．本研究ではグラフ書換え後，既にパターンマッチを行った部分グラフに対して再度パターンマッチを行う回数を減らすことでパターンマッチにかかる実行時間を抑えるアルゴリズムを考え，実際にオーダーが改善されることを確認した．

## 1　はじめに

プログラミングやモデリングの対象には, 代数的データ構造を超えたグラフ構造の表現や, グラフ構造に対するパターンマッチング機能と変換機能が有用となるものも少なくない. このような機能を備えたグラフ書換え言語のうち, CSLMNtal [15] は, ユーザ定義可能な強力なパターンマッチング機能を提供する点を特徴とする.

CSLMNtal では, 組み込みの型に加え, ユーザが再帰的に定義した型を用いてパターンマッチを行うことでグラフを書き換えることができる. しかし素朴な実装では, パターンマッチに成功しグラフを書き換えるたびに, それまでのマッチングで得られた型情報を全て破棄した上で書換え後のグラフについて最初からパターンマッチをやり直していたため, 型情報の再利用による実行時間の改善が課題とされていた.

以上の課題に対し先行研究 [13] では, 特定の例題を取り上げて, グラフ書換え後, すでにパターンマッチを行った部分グラフに対して繰り返しパターンマッチを行う回数を減らすことで, パターンマッチにかかる実行時間を抑えられることを示していた. しかし, そのような効率的なパターンマッチを行う中間言語を生成する汎用的なアルゴリズムは示されていなかった.

そこで本研究では, 与えられた CSLMNtal のプログラムを元にして, 効率的なパターンマッチを行う中間言語の命令列へとコンパイルするアルゴリズムを構築し, 実際に実行時間のオーダーが改善されることを, 様々な例題を用いて確認する.

## 2　CSLMNtal

本節では, 言語モデル LMNtal [9,10] の拡張として設計された言語である CSLMNtal [15] について説明する. なお, 本論文と関連の薄い内容については割愛しているため, LMNtal と共通である 2.3 節までの内容の詳細は [9] を, それ以降の内容については [15] をそれぞれ参照されたい.

---

* Optimization Methods for Iterative Pattern Matching based on Recursive Graph Patterns
This is an unrefereed paper. Copyrights belong to the Authors.

Ryoya Shirai, Ren Imagawa, Naoki Yamamoto, Kazunori Ueda, 早稲田大学基幹理工学研究科情報理工・情報通信専攻, Dept. of Computer Science and Communications Engineering, Graduate School of Fundamental Science and Engineering, Waseda University.

$$
\begin{array}{rcll}
X & ::= & \langle \mathit{LinkName} \rangle & \\
p & ::= & \langle \mathit{AtomName} \rangle & \\
P & ::= & \epsilon & (\text{空}) \\
 & | & p(X_1, \ldots, X_n)\ (n \geq 0) & (\text{アトム}) \\
 & | & P, P & (\text{分子}) \\
 & | & P\ \texttt{:-}\ P & (\text{ルール})
\end{array}
$$

**図 1　CSLMNtal プロセス $P$ の構文**

### 2.1　概要

CSLMNtal のプログラムは, グラフとルール（グラフを書き換えるための規則）の多重集合（**プロセス**）によって書き表される．また, CSLMNtal の最大の特徴は, LMNtal に組み込まれている型に加え, ユーザが再帰的パターンに基づき定義した型を使用し, パターンマッチを行える点である．これにより, 例えばリストや二分木のような代数的データ型を定義でき, それらに穴が空いた差分リスト (list segment) や文脈 (context) のようなグラフ構造の型も定義できる．

CSLMNtal のプロセスの構文を図 1 に示す．ここで $\langle \mathit{LinkName} \rangle$ は大文字アルファベットから始まる識別子, $\langle \mathit{AtomName} \rangle$ はそれ以外から始まる識別子または “’” で囲まれた文字列である．分子を構成する “,” はルールを構成する “:-” よりも強い結合力を持つ．また, プロセス上の構造合同関係 (≡) を図 2 に示す規則を満たす最小の同値関係として定義する．この構造合同関係は, グラフ理論におけるグラフ同型に対応する同値関係であり, 構文上の差異を吸収する．

### 2.2　アトムとリンク

**アトム・リンク**は, それぞれグラフ理論における節点・無向辺と対応しているが, CSLMNtal においてはアトムから伸びるリンク（引数）に全順序が付されている点に特徴がある．この引数の数をアトムの**価数** (arity) と呼ぶ．

リンクは 3 つ以上のアトムを連結することはできない (**グラフのリンク条件**)．従って, 任意のプロセスには同じリンクは高々2 回出現する．あるプロセスで 2 回出現するリンクを**局所リンク** (local link), 1 回出現するリンクを**自由リンク** (free link) と呼ぶ．例えば `a(X), b(X,Y)` について, `X` は局所リンクであり `Y` は自由リンクである．

$$
\begin{array}{rrcl}
1. & \epsilon, P & \equiv & P \\
2. & P_1, P_2 & \equiv & P_2, P_1 \\
3. & P_1, (P_2, P_3) & \equiv & (P_1, P_2), P_3 \\
4. & X = X & \equiv & \epsilon \\
5. & X_1 = X_2 & \equiv & X_2 = X_1 \\
6. & P & \equiv & P[X_1 \leftarrow X_2] \\
7. & X_1 = X_2, P & \equiv & P[X_1 \leftarrow X_2] \\
8. & P_1 \equiv P_1' & \Rightarrow & P_1, P_2 \equiv P_1', P_2
\end{array}
$$

**図 2　CSLMNtal プロセスの構造合同規則**

CSLMNtal で唯一予約されているアトムは 2 価の `'='`（コネクタ）であり, その意味は構造合同規則 4., 5. 及び 7. に記されている．直観的には 2 価の `'='` アトムはその 2 つの引数を直接リンクで接続することを表す．例えば `a(X), b(Y), X = Y` は `a(X), b(X)` と構造合同である．

### 2.3　ルール

ルールの操作的意味論 (→) を図 3 に示す．ルールは 2 つのプロセスを “:-” で結んだものであり, 直観的には “左辺にマッチするプロセスを右辺のプロセスに書き換えてよい” という規則を表す．例えば, プロセス `p(A,A), (p(X,Y) :- q(X,Y))` は次のように書き換えられる．

```
    p(A,A),          (p(X,Y) :- q(X,Y))
≡   p(X,X),          (p(X,Y) :- q(X,Y))
≡   p(X,Y), Y = X,   (p(X,Y) :- q(X,Y))
→   q(X,Y), Y = X,   (p(X,Y) :- q(X,Y))
≡   q(X,X),          (p(X,Y) :- q(X,Y))
```

ルールの適用前後において自由リンクが増減してはならないため, ルール両辺の自由リンクの集合は等しい必要がある（**ルールのリンク条件**）．よって, ルール中に出現するリンクは必ず以下の 4 通りに分類することができる．

$$P_1, (P_1 \ \texttt{:-}\ P_2) \to P_2, (P_1 \ \texttt{:-}\ P_2)$$

$$\frac{P_1 \to P_1'}{P_1, P_2 \to P_1', P_2}$$

$$\frac{P_1 \to P_1' \quad P_1 \equiv P_2 \quad P_1' \equiv P_2'}{P_2 \to P_2'}$$

**図 3　CSLMNtal の操作的意味論**

$$P ::= \cdots \mid \$p[X_1, \ldots, X_n]$$

**図 4　CSLMNtal の拡張構文：プロセス文脈**

1. 左辺に 2 回出現： リンクの削除
2. 右辺に 2 回出現： リンクの生成
3. 両辺に 1 回ずつ出現： リンクの接続先の変更
4. 両辺に 2 回ずつ出現： リンクの保持

### 2.4　プロセス文脈

主に算術演算や不特定多数のアトムの一括複製・削除のために, 形式的定義にある構文・意味論に加えて **プロセス文脈** (process context) と呼ばれる拡張構文が存在する. プロセス文脈の構文を図 4 に示す.

プロセス文脈はアトムに似ているが, ルール中にのみ書くことができる. ルール左辺に書かれたプロセス文脈 $\$p[X_1, \ldots, X_n]$ は, $X_1, \ldots, X_n$ のみを自由リンクにもつ 0 個以上のアトムからなるグラフを代表する. このとき, $X_1, \ldots, X_n$ は互いに異なるリンク名でなければならない. 各プロセス文脈には, そのプロセス文脈がマッチするプロセスの条件を指定する **ガード条件** (guard condition) をルールの **ガード部** (guard) に指定する必要がある. ガード部を持つルールの構文を図 5 に示す. ガード部はルールの “`|`” と “`:-`” の間をいい, ここに任意個のガード条件を書くことができる. ガード条件はあらかじめ処理系で定義された型制約か, ユーザが定義した型制約のいずれかである.

ルール左辺に書かれたプロセス文脈は, 同じ価数・名前のプロセス文脈を右辺にも書くことができ, 右辺に書かれたそれらは書換えの際に左辺でマッチしたプロセスと置き換えられる. その際, プロセス文脈の自由リンクは右辺で指定したものに置き換えられる. またルール左辺には出現しないプロセス文脈をルール右辺に書くことはできない. 加えて, ルール左辺に出現する全てのプロセス文脈名はガード部でそれぞれ 1 回以上出現する必要がある.

$$\begin{array}{rcl} t & ::= & \langle \mathit{TypeName} \rangle \\ c & ::= & t(\$p) \\ P & ::= & \cdots \mid P \mid c_1, \ldots, c_n \ \texttt{:-}\ P \ (n \geq 0) \end{array}$$

**図 5　CSLMNtal の拡張構文：ガード部を持つルール**

### 2.5　型定義

本論文の以下の部分では, $X_1, \ldots, X_n$ $(n \geq 0)$ のような自由リンク名の列を単一の飾り文字で $\mathcal{X}$ などと表現することがある. ある $t$ 型のプロセス文脈 $\$p[\mathcal{X}]$ がプロセス $P$ にマッチすることを $P \in t(\mathcal{X})$ と定義すると, プロセス文脈の型は自由リンク名の列をパラメタに持つプロセスの（無限）集合であるとみなせる. CSLMNtal においてプロセス文脈の型を定義するための構文を図 6 に示す. ただし, 可読性のため右辺が空であるような *TypeRule* の “`:-`” は省略することがある. 便宜上, *TypeRule* の先頭にルールの名前 *RuleName* を付すことがある.

例えば, 図 7 に示す型定義は, 直観的には `abc(X)` が, 3 つのプロセス `a(X)`, `b(X)`, `c(X)` からなる 3 要素の集合であることを言明しており, プロセス $P$ は, $P$ が `a(X)`, `b(X)`, `c(X)` のいずれかであるとき, かつそのときに限り型 `abc` のプロセス文脈にマッチする.

集合の要素を具体的に全て列挙するだけでなく, “`:-`” を用いることで集合を（相互）再帰的に定義することもできる. これを用いて, 例えば差分リスト（とみなせるプロセス）の集合を定義すると図 8 のように表せる. ただし, 差分リストとは空リストで終端されていない線形リストのことをいう. すなわち, `H`, `T` のみを自由リンクとするプロセス $P$ が型 `dlist` を持つプロセス文脈にマッチするのは,

1. $P =$ `H = T`,

$$
\begin{aligned}
\mathit{TypeRule} &::= [\mathit{RuleName}\ \texttt{@@}]\ P\ \texttt{:-}\ t_1(\mathcal{X}_1), \ldots, t_n(\mathcal{X}_n).\ (n \geq 0) \\
\mathit{TypeDef} &::= \texttt{typedef}\ t(\mathcal{X})\ \{\mathit{TypeRule}_1\ \ldots\ \mathit{TypeRule}_n\}\ (n \geq 0)
\end{aligned}
$$

**図 6　型定義の構文**

```
typedef abc(X) { a(X). b(X). c(X). }
```

**図 7　型定義の例**

```
typedef dlist(T, H) {
  H = T.
  H = '.'(A, B) :- int(A), dlist(T, B).
}
```

**図 8　dlist の型定義**

2. $P = \texttt{H = '.'(A,B)},\ P_1,\ P_2$
   where $P_1 \in \texttt{int(A)}, P_2 \in \texttt{dlist(T,B)}$

のいずれかの場合のみである. ただし, `int` は 1 価の整数アトムを表す組み込みの型である. 例えば, 次のプロセスは `dlist(T,H)` の元である:

`H = '.'(1,'.'(2,'.'(3,'.'(4,T))))`

型定義 *TypeDef* は次の構文規則を満たさなければならない:

1. 型 $t(\mathcal{X})$ の引数 $\mathcal{X}$ に出現するリンク名は互いに異なる
2. どの *TypeRule* も, $t(\mathcal{X})$ を左辺に書き加えるとルールのリンク条件を満たす
3. 左辺のあらゆる空でない部分プロセスは少なくとも一つの自由リンクを持たなければならない

形式的には *TypeRule* とグラフの集合との対応付け $S$ を次のように定義する:

$$
\begin{aligned}
&P\ \texttt{:-}\ t_1(\mathcal{X}_1), \ldots, t_n(\mathcal{X}_n) \\
&\qquad \mapsto_S\ \{P, P_1, \ldots, P_n \mid P_k \in t_k(\mathcal{X}_k)\}
\end{aligned}
$$

この $S$ のもとで, 型定義

$\texttt{typedef}\ t(\mathcal{X})\ \{\mathit{TypeRule}_1\ \ldots\ \mathit{TypeRule}_n\}$

は集合 $t(\mathcal{X})$ を

$$S(\mathit{TypeRule}_1) \cup \cdots \cup S(\mathit{TypeRule}_n)$$

と定義するものと決める. $S$ と $t(\mathcal{X})$ の定義が相互再帰的であるが, それぞれ集合の包含関係に関して最小の解を取るものとする.

### 2.6　略記法

可読性, 記述性のために CSLMNtal では次の略記法を認める.

#### 2.6.1　ピリオドの使用

“`:-`” よりも結合力の弱い “`,`” として “`.`” を用いる. これによって, ルールとアトムの分子を作成する際に, ルールを括弧で囲む必要がなくなる. 例えば, 以下の 2 つのプロセスは等価である.

```
a(X), a(X,Y), a(Y). a(X) :- b(X)
a(X), a(X,Y), a(Y), (a(X) :- b(X))
```

#### 2.6.2　引数リストの略記法

アトムの引数リストに対して, 次の 3 つの略記法を適用する.

1. 引数のないアトム `a()` を単に `a` と表記する
2. アトム `a(X,Y)` の最終引数を外に出して `Y = a(X)` と表記する
3. `'='` アトムを含むプロセス `a(X,Y), Y = b` を単に `a(X,B)` と表記する

例えば, 以下の 4 つの表記は全て等価である.

- `a(X,Y), b(X), c(Z,Y)`
- `a(X,Y), X = b, Y = c(Z)`
- `a(b, c(Z))`
- `c(Z) = a(b)`

また, プロセス文脈に対しても同様の略記法を用いる.

#### 2.6.3　リストの略記法

CSLMNtal では慣習的に, cons を表す 3 価の `'.'` アトムと, 空リストを表す 1 価の `'[]'` アトムの組み合わせによってリストを表現する. この方法でリストを書く場合, Prolog 流のリスト記法を用いることができる. 例えば, `X = [a,b]` はリスト `'.'(a,'.'(b,[]),X)` を表し, また `X = [a,b | X]` は循環リスト `'.'(a,'.'(b,X),X)` を表す.

## 3 CSLMNtal 中間命令

本研究では, CSLMNtal プログラムからパターンマッチの順序が効率化された中間言語を生成するアルゴリズムの構築を目的としており, その説明のため, 中間言語を記述するのに用いられる CSLMNtal の中間命令について説明する. 本節は概ね文献 [11–13] に基づくが, 本研究と直接関係しない中間命令については省略する.

### 3.1 概要

CSLMNtal の中間言語では, `Subrule` という中間命令列のブロックが実行の基本単位となる. 各 `Subrule` 内の中間命令は基本的に上から順に実行されるが, 相互に呼び出すことができる点で, 命令形言語におけるサブルーチンに相当する. ただし, マッチングの失敗などにより中間命令が失敗となった場合は, それ以前の命令へとバックトラックして実行を継続する. また, 呼び出しの開始地点となる `Subrule` を特に `Rule` と書く.

各中間命令は, 命令名と引数からなり, 引数リストは角括弧 `[...]` で囲まれたカンマ区切りの列で記述する. 各引数は自然数, 文字列（`"..."` で囲む）, ファンクタ, それらからなるリストのいずれかである. ここで, ファンクタとはアトムの名前と価数の組を指し, 例えば 3 価の `a` アトム `a(X,Y,Z)` のファンクタは `'a'_3` のように記述する. また, 一部の制御構造を担う命令では引数リストの代わりに, 二重の角括弧 `[[...]]` で囲まれた命令列を引数に取ることがある.

ここでは, 主に `Subrule` の呼び出しなど, 中間命令列の実行順を制御する 4 つの中間命令を紹介する.

- `subrule` **命令** 名称の通り他の `Subrule` を呼び出す命令である. この命令は呼び出し元の `Subrule` に配置され, 引数に呼び出し先の `Subrule` の名前をとる.
- `succreturn` **命令** 呼び出し元の `Subrule` に戻る時に用いられる命令である. この命令は呼び出し先の `Subrule` に配置され, 戻った後は呼び出し元の `subrule` 命令を成功扱いとし, 次の命令から実行を続ける.
- `failreturn` **命令** この命令も, 呼び出し元の `Subrule` に戻る時に用いられるが, 戻った後は呼び出し元の `subrule` 命令を失敗扱いとしてバックトラックする.
- `branch` **命令** 命令列を引数に取る命令で, `branch` 命令の内部の命令が成功している間はそのまま branch の中の命令を 1 つずつ実行していき, 失敗すると `branch` 命令から抜け, 直後の命令から実行を続ける.

また図 9 のように, `branch` 命令の内部に `subrule` 命令を配置することで, 呼び出し先の `Subrule` から `succreturn` で戻ってきたときは `subrule` 命令の次の命令（命令列 1）から再開し, `failreturn` で戻ってきたときは `branch` 命令の直後（命令列 2）から再開できる.

```
1 branch[[
2   subrule [.., .., .., ..]
3   (命令列 1)
4 ]]
5 (命令列 2)
```

**図 9 branch 命令と subrule 命令を組み合わせて配置する例**

## 4 問題の分析と提案手法

現行の CSLMNtal コンパイラ †1 が生成する中間命令列は, ルールが繰り返し適用される際, 毎回最初から型の再検査をやり直している. しかし, 再帰的パターンを繰り返し検査すると時間計算量が悪化するため, 再利用可能な情報はできる限り再利用したい.

田口による先行研究 [13] において, 特定の数種類の例題に対しては, 再帰的パターンの繰り返し検査を効率化した中間命令列が与えられたが, そのような中間命令列を生成する汎用的なアルゴリズムは存在しない.

そこで本研究では, CSLMNtal において定義される再帰的パターンの構造にどのような種類が存在するかを特定・分類し, 型検査を効率的に行う中間言語

†1 https://github.com/lmntal/lmntal-compiler

を生成するアルゴリズムを, 分類したそれぞれの種類に対して提案する.

### 4.1 概要

CSLMNtal における書換えは, まずルール左辺とのマッチング（型検査含む）を行い, 成功した場合にルール右辺へと書き換えるという手順が基本であるが, この際にルール左辺と右辺で共通する部分がある場合は, 共通部分はそのまま引き継いで, 差分のみを書き換える. 以下, この実際に書き換えられる部分グラフを, **書換え対象部分**と呼ぶ.

田口による効率化されたパターンマッチは最初に取得した起点となるアトム（**init アトム**と呼ぶ）からリンクを辿って行われる. このとき, 書換え対象部分と init アトムの間に存在するユーザ定義型のプロセス文脈の個数が多いほど, グラフ書換え後のパターンマッチに型検査の情報を再利用できるため効率化されるが, 逆に 1 つも存在しないと効率化はほとんどできない. 従って, init アトムと書換え対象部分の間にプロセス文脈が存在する場合（分類 1）としない場合（分類 2）でまず分類する.

### 4.2 分類 1： init アトムと書換え対象部分の間にプロセス文脈が存在する場合

この場合は, （パターンマッチに成功し）グラフを書き換えた後, 書換え対象部分とパターンマッチする直前に型検査を行ったプロセス文脈 $\mathrm{process}_p$ について最後に *TypeRule* を適用した段階から パターンマッチが再開する. 図 10 を例に説明すると, 赤部分が書き換えられるとき, その直前に型検査を行った青のプロセス文脈が $\mathrm{process}_p$ である. また, $\mathrm{process}_p$ が *TypeRule* を 2 個持ち, 1 段階目で 2 つめの *TypeRule*, 2 段階目で 1 つめの *TypeRule* の順で *TypeRule* を適用したときに, グラフ書換えが行われたと仮定すると, 2 段階目の *TypeRule* の適用からパターンマッチを再開する.

このパターンマッチが全て失敗した後, 既に試したプロセス文脈についてパターンマッチをやり直すか否かで, やり直す場合（**分類 1-1**）とやり直さない場合（**分類 1-2**）にさらに分類することができる.

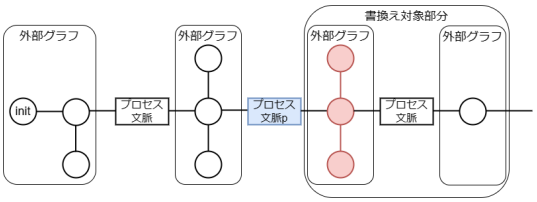


**図 10 分類 1： パターンマッチの例**

この分類は, 書換え対象部分とパターンマッチする直前に型検査を行うプロセス文脈 $\mathrm{process}_p$ の型 $t_p$ の性質に依る. これを説明する上で, まず $t_p$ の *TypeRule* に手を加える. *TypeRule* は再帰のないベースケースと再帰ルールに分類できる. ここではパターンマッチの都合上, ベースケースが先に並んでいるものとする. そして, $t_p$ 型が再帰的 *TypeRule* を複数持つとき, 全ての再帰的 *TypeRule* の再帰部分にベースケースを代入する. この作業により, 後ろの *TypeRule* が先の *TypeRule* のサブグラフとなるとき分類 1-1、ならないときは分類 1-2 とする。なお, 上記で分類 1-1 となる例題のうち、*TypeRule* を適用する順序に意味がない例題の場合 *TypeRule* の順序を入れ替えることで分類 1-2 の例題として取り扱うことができるものもある。

次節からは, 効率的な中間言語生成について,

- 分類 1-1・分類 1-2 に共通する部分,
- 分類 1-1 に特有の部分,
- 分類 1-2 に特有の部分

に分けて順に説明する. 以降, ユーザ定義型のプロセス文脈を含むルール左辺において, プロセス文脈を含まない連結成分をそれぞれ**外部グラフ**と呼ぶことにする.

#### 4.2.1 分類 1-1・1-2 に共通する部分

各プロセス文脈では再帰的にパターンマッチが行われるため, プロセス文脈に関する一連の中間命令列を `Subrule` として独立させる. すなわち, 外部グラフ・プロセス文脈ごとに 1 つの `Subrule` を立てて, その中にパターンマッチに関する中間命令列を配置していく.

**手順 (i)［init アトムの取得と外部グラフのパターンマッチ］**（図 11, 図 12） ここで init アトムの候

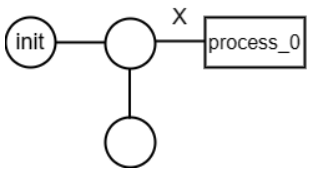


**図 11　分類 1： 手順 (i) のパターンマッチ図**

```
1 Compiled Subrule @t0
2   spec
3   (パターンマッチに関する命令列)
4   subrule
5   proceed / failreturn
```

**図 12　分類 1： 手順 (i) の中間命令列の構造**

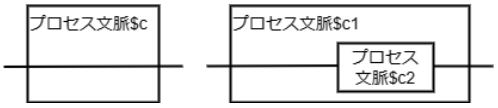


**図 13　分類 1： 手順 (ii-i) の TypeRule**

補となるのは, はじめに型検査の対象となるプロセス文脈 $\mathrm{process}_0$ のリンクのうちベースケース以外の *TypeRule* の左辺に登場するリンクに接続されている外部グラフ $E$ のアトムである. 次にルールの左辺を参照してリンクを辿ることで, $E$ のパターンマッチを行う. そして, プロセス文脈 $\mathrm{process}_0$ と連結しているリンク ($X$ とする) を取得し, $\mathrm{process}_0$ のパターンマッチを行う `Subrule` を呼び出す.

**手順 (ii)［プロセス文脈 $\mathrm{process}_0$ の型検査］**（図 15）プロセス文脈のパターンマッチは, 1 つめの *TypeRule* から順に適用していく. 各 *TypeRule* の適用に関しては外部グラフのパターンマッチと同様, $\mathrm{process}_0$ の型 $t_0$ の各 *TypeRule* の左辺に従ってリンク $X$ よりパターンマッチを行う. このとき中間命令列には, 各 *TypeRule* と対応する `branch` を配置し, それぞれのパターンマッチに関する中間命令列を各 `branch` の中に記述する.

**手順 (ii-1)** プロセス文脈の内部に, 他にプロセス文脈が何もないケースだけでなく, ベースケースのある他の型を持つプロセス文脈が内部に存在するケースもベースケースとする（図 13）.

ベースケースの *TypeRule* では, パターンマッチ後

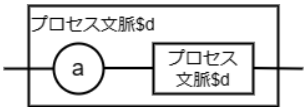


**図 14　分類 1： 手順 (ii-2) の TypeRule**

```
1  Compiled Subrule @t1
2    spec
3    branch[[
4      (パターンマッチに関する命令列)
5      subrule [.., .., .., ..]
6    ]]
7    ...
8    branch[[
9      (パターンマッチに関する命令列)
10     subrule [.., .., .., ..]
11   ]]
12   failreturn
```

**図 15　分類 1： 手順 (ii) の中間命令列の構造**

$\mathrm{process}_0$ と連結している次の外部グラフのパターンマッチに関する `Subrule` を呼び出す. なお本論文が対象とするプログラムは, ベースケースの *TypeRule* は型検査に投げられたリンクが直接繋がったものと仮定する. よって, この *TypeRule* は辿ってきたリンクを次の外部グラフのパターンマッチを行う `Subrule` にそのまま受け渡す.

**手順 (ii-2)** プロセス文脈の中に同じ型定義を持つプロセス文脈が含まれている場合を再帰ケースとする（図 14）.

再帰的 *TypeRule* では, *TypeRule* がユーザ定義型のプロセス文脈を含むことになるが, そのプロセス文脈の型制約が $t_0$ 型と等しいか異なるかで中間命令列の配置が異なる. 等しい場合, パターンマッチ後, 大元の `Subrule` を再帰的に呼び出す. 異なる場合, その型制約を検査する新たな `Subrule` を同様に記述し, `subrule` 命令により呼び出す.

**手順 (ii-3)** 全ての *TypeRule* の適用が失敗した場合, 呼出し元へ `failreturn` する.

**手順 (iii)　［マッチングの繰り返し］**　この後, 外部グラフとプロセス文脈のパターンマッチを交互に行っていくが, これ以降のパターンマッチに関する中間命令列の配置は $\mathrm{process}_p$ より前か後ろかで異なる.

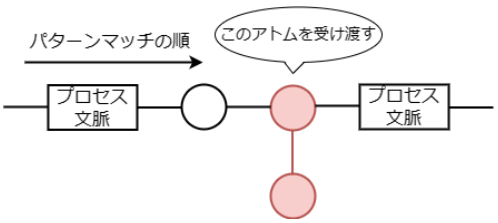


**図 16　分類 1： 手順 (iii) で受け渡すアトム**

$\text{process}_p$ の型検査に関する中間命令列の配置は後の節で説明する．また, グラフの書換えが行われる予定の外部グラフについては, その外部グラフの中で最初に辿ったアトムを最後の Subrule まで subrule 命令により受け渡す（図 16）．

**手順 (iii-1)［$\text{process}_p$ より前のプロセス文脈・外部グラフに関する中間命令列］** このとき, パターンマッチに関する中間命令列の配置は上記と同様である．

**手順 (iii-2)［$\text{process}_p$ より後ろのプロセス文脈・外部グラフに関する中間命令列］**（図 17, 図 18） このとき, パターンマッチが成功しグラフが書き換えられると, $\text{process}_p$ の型検査まで制御を戻す必要がある．よって, branch 命令の中の subrule 命令の直後に succreturn 命令を配置する（図 17, 5, 11 行目）．外部グラフに関する中間命令列では subrule 命令を branch 命令の中に配置してから succreturn 命令を配置する（図 18, 6 行目）．これにより, 呼出し先から succreturn で戻ってきたらそのまま呼出し元へ succreturn され, failreturn で戻ってきたら branch を抜け branch 命令の後の命令を実行する．

**手順 (iv)［グラフの書換え］**（図 19） パターンマッチが成功した後, グラフ書換えに関する中間命令列を配置する．なお, 現行のコンパイラではプロセス文脈の自由リンクがルール左辺の自由リンクとなることを許していないため, 本研究でもそれに従う．よって, パターンマッチに関する中間命令列がプロセス文脈に関する Subrule で終了することはなく, 必ず外部グラフに関する Subrule で記述が終了する．この外部グラフの Subrule の中にグラフ書換えに関する中間命令列を続けて配置する．グラフ書換えに関する具体的な中間命令列の説明は本論文では割愛するが, 最後の Subrule まで受け渡されてきたアトムからリンクを辿って書き換えるアトムを再取得し, リンクを付け替えることでグラフが書き換えられる．1 回以上グラフを書き換えたという情報を保持するため, 最後に succreturn 命令を配置する．

```
1  Compiled Subrule @t3
2    spec
3    branch[[
4      subrule [.., .., .., ..]
5      succreturn []
6    ]]
7    ...
8    branch[[
9      (パターンマッチに関する命令列)
10     subrule [.., .., "t3", ..]
11     succreturn []
12   ]]
13   failreturn []
```

**図 17　分類 1-1： $\text{process}_p$ より後ろのプロセス文脈に関する命令列**

```
1  Compiled Subrule @t4
2    spec
3    (パターンマッチに関する命令列)
4    branch[[
5      subrule [.., .., "t5", ..]
6      succreturn []
7    ]]
8    failreturn []
```

**図 18　分類 1-1： $\text{process}_p$ より後ろの外部グラフに関する命令列**

```
1  Compiled Subrule @tn
2    spec
3    (パターンマッチに関する命令列)
4    (グラフ書換えに関する命令列)
5    succreturn
```

**図 19　分類 1： 手順 (iv) の中間命令列の構造**

#### 4. 2. 2　分類 1-1 に特有の部分（図 20）

分類 1-1 では, グラフ書換え後, 書換え対象部分とパターンマッチする直前に型検査を行った $\text{process}_p$ について, 今まで適用した *TypeRule* をやり直す．このとき, リンクの再取得を行うため, $\text{process}_p$ に関する中間命令列を書き換える．

ここでは, $\text{process}_p$ とその次の外部グラフとを連結するリンクを再取得するため, 各 *TypeRule* においてこのリンクを取得する命令列を Subrule として独立

```
1  Compiled Subrule @t1
2    spec
3    branch[[
4      subrule [.., .., .., ..]
5      succreturn []
6    ]]
7    ...
8    branch[[
9      (パターンマッチに関する命令列)
10     subrule [.., .., "t2", ..]
11     succreturn []
12   ]]
13   failreturn []
14
15 Compiled Subrule @t2
16   spec
17   (リンクを取得する命令列)
18   subrule [.., .., "t1", ..]
19   branch [[
20       subrule [.., .., "t2", ..]
21       succreturn []
22   ]]
23   succreturn []
```

**図 20　分類 1-1：** $\mathrm{process}_p$ **に関する命令列**

```
1  Compiled Subrule @t1
2    spec
3    branch[[
4      subrule [.., .., .., ..]
5      succreturn []
6    ]]
7    ...
8    branch[[
9      (パターンマッチに関する命令列)
10     subrule [.., .., "t2", ..]
11     succreturn []
12   ]]
13   failreturn []
14
15 Compiled Subrule @t2
16   spec
17   (リンクを取得する命令列)
18   subrule [.., .., "t1", ..]
19   subrule [.., .., "t2", ..]
20   failreturn []
```

**図 21　分類 1-2：** $\mathrm{process}_p$ **に関する命令列**

させ再帰的に呼び出す（10, 20 行目）. なお, 本論文が対象とするプログラムは, ベースケースの *TypeRule* は自由リンク同士を直接繋げるものと仮定する. このため, 新たにリンクを取得する命令列は存在せず, 独立させる必要もない. また, この Subrule を必ず succreturn 命令（23 行目）で戻すことで, 今まで適用した *TypeRule* も含めてやり直すことになる.

#### 4.2.3　分類 1-2 に特有の部分（図 21）

分類 1-2 では, グラフ書換え後 $\mathrm{process}_p$ について今まで適用した *TypeRule* をやり直さない. このとき, 分類 1-1 と同様リンクの再取得を行うため, $\mathrm{process}_p$ を書き換える. 分類 1-1 と異なる部分はリンクを取得する Subrule の最後に failreturn 命令を配置する点である.（20 行目）. これにより, $\mathrm{process}_p$ にこの段階で適用可能な全ての *TypeRule* の適用が失敗したとき, 1 段階前の *TypeRule* の適用に failreturn で戻ることで, $\mathrm{process}_p$ にまだ適用したことのない *TypeRule* のみをやり直すことができる.

### 4.3　分類 2：init アトムとグラフの書換え部分の間にプロセス文脈が存在しない場合（図 22）

この場合は, グラフ書換え後, 全てのユーザ定義型のプロセス文脈について初めから型検査を行うことになるため, 効率化が働かない. しかし, 書き換えられた後のグラフについて, プロセス文脈にあたる部分が必ずユーザが定義した型になっているということが事前にわかっている例題では, 書き換えられた部分のパターンマッチのみを再度行えば十分である.

まず, 1 回目のパターンマッチが成功後, succreturn 命令で init アトムを取得する Subrule まで戻ってくるよう subrule 命令の直後に succreturn 命令を配置する. 次に, グラフ書換えの命令列を配置した Subrule を呼び出し, 書き終えたら succreturn 命令で戻る. 最後に, 書き換えられた部分について再度パターンマッチを行う Subrule を呼び出す (t3). なお, グラフ書換えの Subrule に関しては再利用できるため, Subrule として独立させておく (t4).

## 5　例題

本節では, 例題を用いてパターンマッチの手順や中間命令列の配置などを簡単に説明する.

### 5.1　分類 1 の例

#### 5.1.1　分類 1-1 の例：評価文脈による計算順序の制御

本例題は, 戻り掛け順に計算が進んでいく例として文献 [15] に挙げられている. プログラムを図 23 に

```
// 全ての Subrule の呼出し元
Compiled Subrule @t1
  spec
  (パターンマッチに関する中間命令列)
  subrule [.., .., "t2", ..] // パターンマッチ
  subrule [.., .., "t3", ..] // グラフ書換え
  subrule [.., .., "t4", ..] // 書換え後
  proceed []

// 書換え後のパターンマッチ
Compiled Subrule @t4
  spec
  (パターンマッチを行う中間命令列)
  subrule [.., .., "t3", ..]
  subrule [.., .., "t4", ..]
  failreturn []
```

**図 22　分類 2： 中間命令列の構造**

```
typedef context(Hole, Root) {
  c0@@ Root = Hole.
  c1@@ add(X, Y, Root) :-
    int(X), context(Hole, Y).
  c2@@ add(X, Y, Root) :-
    ground(Y), context(Hole, X).
}.

eval(H1), $c[R, H1], add($x, $y, R) :-
  $z = $x + $y, context($c) |
  eval(H1), $c[R, H1], R = $z.
```

**図 23　評価文脈による計算順序の制御プログラム**

示す.

本例題で定義されている型 context は 3 つの *TypeRule* c0, c1, c2 からなり, 再帰ルールの再帰部分にベースケース c0 を代入した時、c2 が c1 のサブグラフとなる. また, 計算順序を制御するために *TypeRule* の適用の順序が重要であるため, *TypeRule* の順序を入れ替えることができない. よって, 分類 1 の例となる.

まず, パターンマッチの順序について説明する. 最初に eval アトムを取得した後, リンク H1 を辿り, プロセス文脈 $c が context 型であるか検査する. そして, context 型のベースケースである c0 から順に $c に適用する. その後, $c に連結する add($x,$y,R) のパターンマッチを行い, ルール左辺全体のパターンマッチを終える.

パターンマッチに成功した場合, 今まで適用を試した *TypeRule* について, 適用したのが新しい順に再度マッチングを試行する. パターンマッチに失敗した場合は, それ以前の *TypeRule* へとバックトラックしていく.

次に, 中間命令列（付録 A.1 節）の配置について説明する. 本例題ではグラフ書換え後, $c に context 型の各 *TypeRule* の適用を行う前に自由リンク Root の再取得を行う必要がある. よって, Root を再取得してから各 *TypeRule* を適用する命令列を Subrule として独立させ, 再帰的に呼び出すようにコードを変換する. 以上から, 中間命令列は以下の 6 つから構成される.

1. eval アトムのパターンマッチを行って @context を呼び出す Rule
2. Root を再取得してから *TypeRule* c0 を適用する Subrule @context
3. context のすべての *TypeRule* を順に適用する Subrule @context_0
4. Root を再取得してから *TypeRule* c1 を適用する Subrule @context_1
5. Root を再取得してから *TypeRule* c2 を適用する Subrule @context_2
6. add($x, $y, R) のパターンマッチ, グラフの書換えを行う Subrule @context_3

各 Subrule 内の中間命令列の配置は, 4 節の分類 1-1 にしたがう. 1, 6 はパターンマッチに関する命令列, 3 は *TypeRule* の適用に関する命令列, 2, 4, 5 はリンクの再取得に関する命令列である.

#### 5.1.2　分類 1-2 の例： スキップリストの要素の削除

線形リストに途中の幾つかのノードを通過するようなエッジを追加したデータ構造をスキップリスト [8] という. ここでは, 2 レベルのスキップリストを考え, 以下では路線図になぞらえて通過しないノードを「急行停車駅」, 通過するノードを「通過駅」と呼ぶ.

例題として, スキップリストの始点を listp, 終点を '[]', 急行停車駅を a, 通過駅を b として, b を全て削除することを考える. 話を簡単にするために, 図 24 のように, 1 つ目の急行停車駅より後の通過駅をすべて削除することとする.

```
typedef skiplist_sub(R1, R2){
  ss@@ R1 = R2.
}.
typedef skiplist(R1, R2, R3, R4){
  s0@@ R2 = R3 :- skiplist_sub(R1, R4).
  s1@@ a(A, B, C, R3, R4) :-
    int(C), skiplist(R1, R2, B, A).
  s2@@ b(A, B, R4) :-
    int(B), skiplist(R1, R2, R3, A).
}.
typedef blist(R1, R2){
  b0@@ R2 = R1.
  b1@@ R2 = b(A, B) :-
    int(B), blist(R1, A).
}.

listp(H1, T1), $s1[H2, T2, T1, H1],
a(H3, T3, $m1, T2, H2), $s2[H4, H3],
b(H5, $m2, H4) :-
  skiplist($s1), int($m1),
  int($m2), blist($s2) |
  listp(H1, T1), $s1[H2, T2, T1, H1],
  a(H3, T3, $m1, T2, H2), $s2[H5, H3].
```

**図 24　スキップリストの通過駅削除プログラム**

まず, パターンマッチの順序について説明する. 最初に listp アトムを取得した後, リンク H1 を辿り, プロセス文脈 $s1 が skiplist 型であるか検査する. skiplist の型定義は 3 つの *TypeRule* s0, s1, s2 からなり, ベースケース s0 から順に $s1 に適用する. その後, $s1 に連結する a アトムのパターンマッチを行う. そして a アトムに連結するプロセス文脈 $s2 が blist 型であるか検査する. blist の型定義は 2 つの *TypeRule* b0, b1 を持ち, ベースケース b0 から順に $s2 に適用する. 最後に $s2 に連結する b アトムのパターンマッチを行い, ルール左辺全体のパターンマッチを終える.

パターンマッチに成功するとグラフ書換えを行い, $s2 に *TypeRule* を適用する最後の段階について b0 を再度適用する所からパターンマッチを再開する. パターンマッチに失敗するとバックトラックを行う.

次に, 中間命令列（付録 A. 2 節）の配置について説明する. 本例題ではグラフ書換え後, $s2 に blist の各 *TypeRule* の適用を行う前に自由リンク R2 の再取得を行う必要がある. よって, R2 を再取得してから各 *TypeRule* を適用する命令列を Subrule として独立させ, 再帰的に呼び出す. また, s0 の右辺で登場する型 skiplist_sub の *TypeRule* ss に関する Subrule も必要である. よって, 中間命令列は以下の 8 つから構成される.

```
typedef dlist(R1, R2){
  d0@@ R2 = R1.
  d1@@ R2 = '.'($a, A) :-
    int($a), dlist(R1, A).
}.

list1(H1), $d1[H2, H1], '[]'(H2),
list2(H3), $d2[H4, H3], '[]'(H4) :-
  dlist($d1), dlist($d2) |
  list1(H1), $d1[H2, H1],
  $d2[H4, H2], '[]'(H4).
```

**図 25　差分リストの連結プログラム**

1. list アトムのパターンマッチを行う Rule
2. skiplist のすべての *TypeRule* を順に適用する Subrule @skiplist
3. skiplist_sub の *TypeRule* ss を適用する Subrule @skiplist_0
4. a アトムのパターンマッチを行う Subrule @skiplist_1
5. R2 を再取得してから *TypeRule* b0 を適用する Subrule @skiplist_2
6. blist のすべての *TypeRule* を順に適用する Subrule @skiplist_3
7. R2 を再取得してから *TypeRule* b1 を適用する Subrule @skiplist_4
8. b アトムのパターンマッチ, グラフの書き換えを行う Subrule @skiplist_5

各 Subrule 内の中間命令列の配置は, 4 節の分類 1-2 にしたがう. 1, 4, 8 はパターンマッチに関する命令列, 2, 3, 6 は *TypeRule* の適用に関する命令列, 5, 7 はリンクの再取得に関する命令列である.

#### 5. 1. 3　分類 1 で複数の連結グラフにパターンマッチする例： 差分リストの連結

差分リストの連結を表すプログラムを図 25 に示す.

まず, パターンマッチの順序について説明する. 最初に 1 つ目のリストのパターンマッチを行う. list1 アトムを取得した後, リンク H1 を辿り, プロセス文脈 $d1 が dlist 型であるか検査する. dlist の型定義は 2 つの *TypeRule* d0, d1 からなり, ベースケース d0 から順に $d1 に適用する. その後, $d1 に繋が

る ’[]’ アトムのパターンマッチを行い, 1 つ目のリストのパターンマッチを終了する.

次に 2 つ目のリストのパターンマッチを, 1 つ目のリストと同様に行う. list2 アトムを取得した後, リンク H3 を辿り, プロセス文脈 $d2 が dlist 型であるか検査する. その後, $d2 に繋がる ’[]’ アトムのパターンマッチを行い, 2 つ目のリストのパターンマッチを終了する.

パターンマッチに成功するとグラフ書換えを行い, $d1 に *TypeRule* を適用する最後の段階について d0 を再度適用する所からパターンマッチを再開する. パターンマッチに失敗するとバックトラックを行う.

次に, 中間命令列の配置（付録 A. 3 節）について説明する. 本例題ではグラフ書換え後, $d1 に dlist の各 *TypeRule* の適用を行う前に自由リンク R2 の再取得を行う必要がある. よって, R2 を再取得してから各 *TypeRule* を適用する命令列を Subrule として独立させ, 再帰的に呼び出す. よって, 中間命令列は以下の 8 つから構成される.

1. list1 アトムのパターンマッチを行って Subrule @dlist を呼び出す Rule
2. R2 を再取得してから $d1 に *TypeRule* d0 を適用する Subrule @dlist
3. $d1 に dlist のすべての *TypeRule* を順に適用する Subrule @dlist_0
4. R2 を再取得してから $d1 に *TypeRule* d1 を適用する Subrule @dlist_1
5. ’[]’ アトムのパターンマッチを行って @dlist_3 を呼び出す Subrule @dlist_2
6. list2 アトムのパターンマッチを行って @dlist_4 を呼び出す Subrule @dlist_3
7. $d2 に dlist のすべての *TypeRule* を順に適用する Subrule @dlist_4
8. ’[]’ アトムのパターンマッチ, グラフの書換えを行う Subrule @dlist_5

各 Subrule 内の中間命令列の配置は, 4 節の分類 1-2 にしたがう. 1, 5, 6, 8 はパターンマッチに関する命令列, 3, 7 は *TypeRule* の適用に関する命令列, 2, 4 はリンクの再取得に関する命令列である. なお, 5, 6, 7 に関してはパターンマッチが成功してグラフが書き換えられると, 4 まで制御を戻す命令列となっている.

```
1 typedef dlist2(R) {
2   R = [].
3   R = ’.’($a, A) :- unary($a), dlist2(A).
4 }.
5
6 list(H1), ’.’($m, H2, H1), $d[H2] :-
7   dlist2($d), int($m) |
8   list(H1), $d[H1].
```

**図 26　差分リストの先頭の要素を続けて削除する**

### 5. 2　分類 2 の例： 差分リストの先頭の要素を続けて削除する例題

例題を表すプログラムを図 26 に示す.

まず, パターンマッチの順序について説明する. 最初に list アトムを取得した後, リンク H1 を辿り, ’.’($n,H2,H1) アトムのパターンマッチを行う. そして, リンク H2 を辿りプロセス文脈 $d が dlist 型であるか検査する. dlist の型定義および検査手順は, 前節と同様である. その後, $d に繋がる ’[]’ アトムのパターンマッチを行い, ルール左辺全体のパターンマッチを終える.

パターンマッチに成功するとグラフ書換えを行い, 再度 list アトムからリンクを辿ってパターンマッチを再開する. パターンマッチに失敗するとバックトラックを行う.

次に, 中間命令列の配置（付録 A. 4 節）について説明する. 本例題ではグラフ書換え後, list アトムからパターンマッチを再開する命令列を新たに Subrule として独立させる. よって, 中間命令列は以下の 5 つから構成される.

1. list アトム, ’.’ アトムのパターンマッチを行う Rule
2. dlist の *TypeRule* の適用を行う Subrule @dlist
3. ’[]’ アトムのパターンマッチを行う Subrule @dlist_0
4. グラフの書換えを行う Subrule @dlist_1
5. ’.’ アトムのパターンマッチを再度行う Subrule @dlist_2

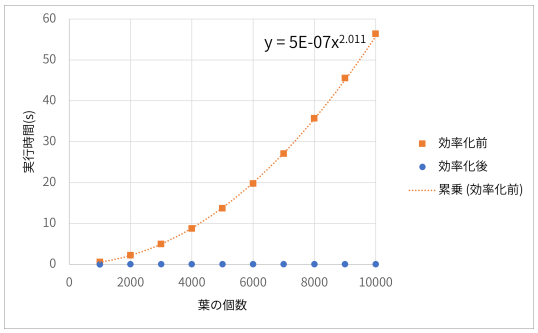


**図 27　左結合的な計算式の個数と実行時間との関係**

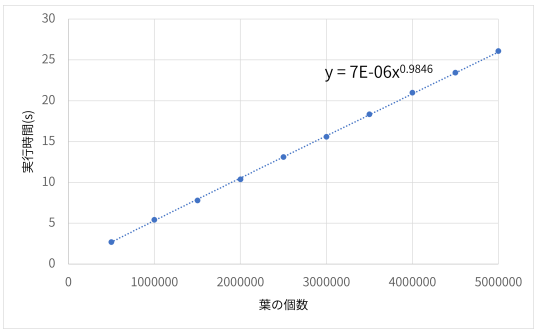


**図 28　左結合的な計算式の個数と実行時間との関係： 効率化済み**

## 6　実験

本節では, 5 節で説明した各例題について, 効率化前と効率化後の実行時間の計測・比較を行う. 効率化前のプログラムとして, 5 節で説明した現行のコンパイラが生成する中間言語を用いる. また, 全ての例題について表 1 の測定環境で実行時間の測定を行った.

**表 1　測定環境**

| | |
|---|---|
| OS: | Ubuntu 20.04.4 LTS |
| CPU: | Intel(R) Core(TM) i7-8550U @ 1.80GHz |
| RAM: | 8GB |

### 6.1　分類 1-1 の例：評価文脈による計算順序の制御

本例題では, add の右オペランドが全て 0（左結合）として, 計測を行う. 加算する数値を全て 0 で統一することで計算による負荷を極力減らしている. 計測結果を図 27 に示す.

効率化後のデータについて全てのデータが 0 付近に集中してしまい, 上記グラフからオーダーが確認できなかった. そこで, 0 の個数を 500,000 から 1,000,000 まで 500,000 ずつ増やして再計測を行った. 計測結果を図 28 に示す.

図 27, 図 28 より, オーダーが $O(n^2)$ から $O(n)$ に落ちることが確認できた.

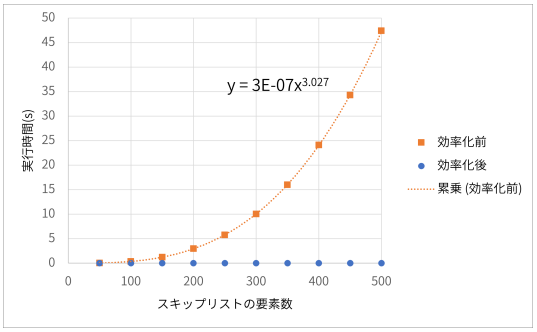


**図 29　スキップリストの要素の個数と実行時間との関係**

### 6.2　分類 1-2 の例：スキップリストの各駅の削除

本例題では, 急行停車駅 a を 10 個と固定し, 各 a の後ろに各駅停車駅 b を同じ個数ずつ並べ, リストの要素の個数の合計を 50 から 500 まで 50 ずつ増やしていき計測を行った. 計測結果を図 29 に示す.

効率化後のデータについて全てのデータが 0 付近に集中してしまい, 上記グラフからオーダーが確認できなかった. そこで, リストの要素の総数を 1,000 から 10,000 まで 1,000 ずつ増やして再計測を行った.. 計測結果を図 30 に示す.

図 29, 図 30 より, オーダーが $O(n^3)$ から $O(n)$ に落ちることが確認できた.

### 6.3　分類 1 で複数の連結グラフを行うパターンマッチの例：差分リストの連結

本例題では, リストの要素数を 1 個に固定し, リストの個数を 1,000 から 10,000 まで 1,000 ずつ増やし

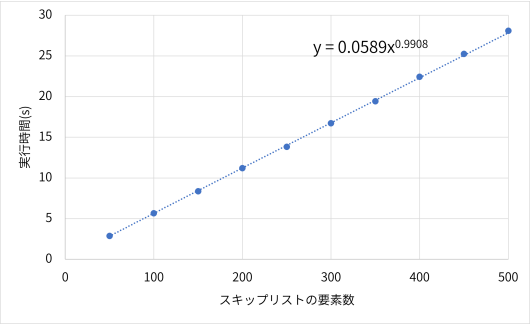


**図 30　スキップリストの要素の個数と実行時間との関係：効率化済み**

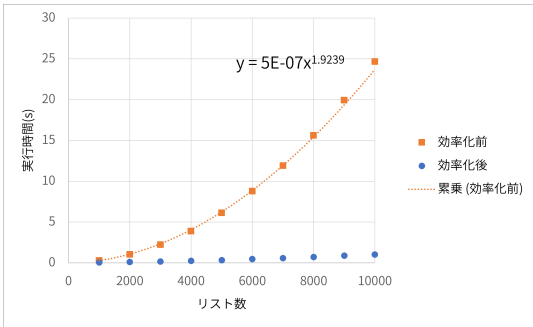


**図 31　差分リストの個数と実行時間との関係**

て計測を行った．計測結果を図 31 に示す．

効率化後のデータについて全てのデータが 0 付近に集中してしまい，上記グラフからオーダーが確認できなかった．そこで，リストの要素の総数を 6,000 から 60,000 まで 6,000 ずつ増やして再計測を行った．計測結果を図 32 に示す．

図 31 より，効率化後のデータについて，効率化が働いていることが確認できる．しかし，図 31 や図 32 において，要素数が多い箇所だと近似曲線を大きく上回っており，多項式では説明できない要因がある．この要因の分析については今後の課題とする．

### 6.4　分類 2 の例：差分リストの先頭の要素を削除する例題

本例題では，リストの要素数を 1,000 から 10,000 まで 1,000 ずつ増やして計測を行った．計測結果を図 33 に示す．

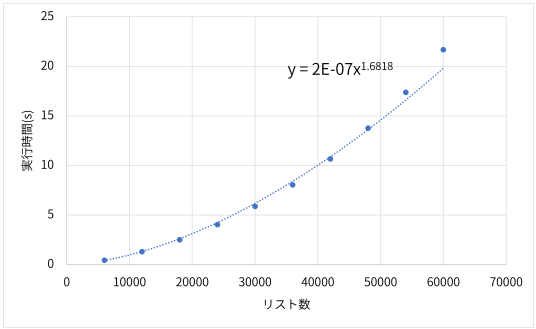


**図 32　差分リストの個数と実行時間との関係：効率化済み**

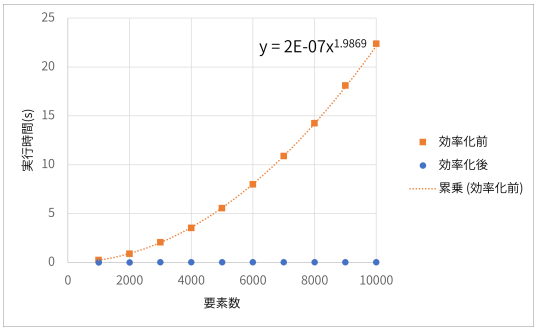


**図 33　差分リストの要素数と実行時間との関係**

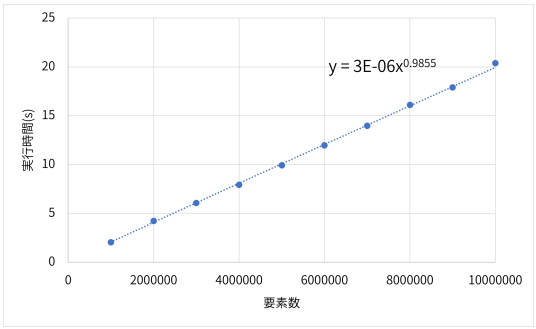


**図 34　差分リストの要素数と実行時間との関係：効率化済み**

効率化後のデータについて全てのデータが 0 付近に集中してしまい，上記グラフからオーダーが確認できなかった．そこで，リストの要素数を 1,000,000 から 10,000,000 まで 1,000,000 ずつ増やして再計測を行った．計測結果を図 34 に示す．

図 33, 図 34 より，オーダーが $O(n^2)$ から $O(n)$ に

落ちることが確認できた.

## 7 関連研究

本研究では, [15] [14] で最初に提案された再帰的グラフパターンマッチングの反復実行の効率化手法を用いて, CSLMNtal の効率的な中間命令列を生成するアルゴリズムについて検討してきた.

CSLMNtal 言語以外の分野においても, グラフパターンマッチングの最適化についての関連研究はいくつか見られる. データベース分野においては, Incremental Graph Pattern Matching [4] が挙げられる. 周期的なパターンやバッチ更新に対する効率的なインクリメンタルアルゴリズムや, 補助的なデータ構造を維持するインクリメンタルアルゴリズムの開発により, グラフが更新されたときに効率的にパターンマッチを行うことができたことが示されている.

グラフ変換 (graph transformation) の分野では, 一般化された推移的閉包を効率的に計算するための複数のアルゴリズム [1] も提案されており, これにより単純なグラフ辺で定義される到達可能領域だけでなく, グラフパターンで定義される複雑な二項関係も計算でき, 幅広いモデリング問題に適用することができるとしている. また [5] では, RETE に基づくインクリメンタルパターンマッチングアルゴリズムを, グラフ変換ツール Groove の網羅的状態空間探索に用いている.

グラフ書換えにおけるインクリメンタルな計算は Kappa グラフでも検討されている [2]. Kappa グラフは, 我々が本文で書き換え対象として取り扱った CSLMNtal グラフと似たグラフ構造を用いているが, 生物系のモデリングを目的として確率的なシミュレーションを行う点が大きく異なる.

プログラミング言語 Egison [3] は, 様々なデータ型に対する強力なパターンマッチングに着眼しており, 使用する例題の種類には共通部分が多いが, 我々の研究はパターンマッチングに基づくデータ構造の反復書換えも同時に考慮している点が異なる.

我々の研究と関数型言語の context pattern [6, 7] は, 通常のプログラミング言語よりも強力なパターンマッチング機能を提供する点や, パターンマッチングをデータ構造の同じ部分に繰り返し適用することを避ける方法を検討している点で似ているが, ユーザ定義可能な再帰パターンを用いるか高階関数を用いるかという点で異なる.

## 8 まとめと今後の課題

### 8.1 まとめ

再帰的に定義されたパターンを用いたグラフ書換えの過程において, パターンの反復利用におけるマッチング作業の重複を避けるための手法を定式化して評価した. 奈良の効率化手法 [14, 15] や, 田口の効率化手法 [13] を踏まえて, CSLMNtal プログラムを効率化された中間命令列へと変換する一般的手法について議論した. その結果, ルール左辺に登場するユーザ定義型のプロセス文脈が 1 個である例題に関しては, 効率化を施した中間命令列において, 効率化前と同等かそれ以上に実行時間のオーダーを落とせることが確認できた.

### 8.2 今後の課題

まず, ユーザ定義型のプロセス文脈より奥を書き換えるルールのための今後の課題について説明する. 現状のアルゴリズムでは, パターンマッチに成功しグラフが書き換えられた後, 書き換えられたアトムの内最初に辿ったアトムまで制御を戻す必要があり, この際に制御を戻した分は情報を捨ててしまっている. 例えば, 差分リストの連結の例題では, 1 個目のリストと 2 個目のリストのパターンマッチを終え連結した後, 元々1 個目のリストの末尾であったアトムの直前まで戻り, リンクを再取得しパターンマッチを再開する. このとき, 元々2 個目のリストであったグラフについて再度プロセス文脈のパターンマッチをする必要がある.

次に, ユーザ定義型のプロセス文脈より手前を書き換えるルールのための今後の課題について説明する. 本研究では, 書き換えられた後のグラフについて, プロセス文脈にあたる部分がユーザが定義した型になっているとみなして, アルゴリズムを考えたが, 本来はこのプロセス文脈にあたる部分について再度型検査を行う必要がある. 例えば, 差分リストの先頭から削

除していく例題では, グラフ書き換え後, 型検査し終えたプロセス文脈について先頭のアトムを取り出した後のグラフも差分リストとなることを前提とした.

この 2 つの課題について, どちらの場合もルールや *TypeRule* から, グラフ書き換え後に最低限パターンマッチをしなければならない部分を導出ができる可能性があると考えているが, そのアルゴリズム化は今後の課題である.

**謝辞** LMNtal の処理系の開発および保守を行っている上田研究室言語班の皆さまに感謝します. 特に CSLMNtal に関する研究について様々な助言を頂いた山田啓太氏, 関連研究の調査のサポートをして頂いた田久健人氏に感謝します. 本研究の一部は，科学研究費補助金 (23K11057)，早稲田大学特定課題研究費 (2023C-428) の補助を得て実施しました.

## 参 考 文 献

[1] Bergmann, G., Ráth, I., Szabó, T., Torrini, P., and Varró, D.: Incremental Pattern Matching for the Efficient Computation of Transitive Closure, *Proc. ICGT 2012*, LNCS, Vol. 7562, Springer, Berlin, Heidelberg, 2012, pp. 386–400.

[2] Boutillier, P., Ehrhard, T., and Krivine, J.: Incremental Update for Graph Rewriting, *Proc. ESOP 2017*, LNCS, Vol. 10201, Berlin, Heidelberg, Springer, 2017, pp. 201–228.

[3] Egi, S. and Nishiwaki, Y.: Non-linear Pattern Matching with Backtracking for Non-free Data Types, *Proc. APLAS 2018*, LNCS, Vol. 11275, Cham, Springer, 2018, pp. 3–23.

[4] Fan, W., Wang, X., and Wu, Y.: Incremental Graph Pattern Matching, *ACM Trans. Database Syst.*, Vol. 38, No. 3(2013).

[5] Ghamarian, A., Jalali, A., and Rensink, A.: Incremental Pattern Matching in Graph-Based State Space Exploration, *Proc. GraBaTs 2010*, ECEASST, Vol. 32, 2010.

[6] Mohnen, M.: Context patterns in Haskell, *Proc. IFL 1997*, LNCS, Vol. 1268, Berlin, Heidelberg, Springer, 1997, pp. 41–57.

[7] Mohnen, M.: Context patterns, part II, *Proc. IFL 1998*, LNCS, Vol. 1467, Berlin, Heidelberg, Springer, 1998, pp. 338–357.

[8] Pugh, W.: Skip lists: A probabilistic alternative to balanced trees, *Commun. ACM*, Vol. 33, No. 6(1990), pp. 668–676.

[9] Ueda, K.: LMNtal as a hierarchical logic programming language, *Theoretical Computer Science*, Vol. 410, No. 46(2009), pp. 4784–4800.

[10] 上田和紀, 加藤紀夫: 言語モデル LMNtal, コンピュータ ソフトウェア, Vol. 21, No. 2(2004), pp. 126–142.

[11] 石川力, 堀泰祐, 村山敬, 岡部亮, 上田和紀: 軽量な LMNtal 実行処理系 SLIM の設計と実装, 情報処理学会第 *70* 回全国大会, (2008).

[12] 村山敬, 工藤晋太郎, 櫻井健, 水野謙, 加藤紀夫, 上田和紀: 階層グラフ書換え言語 LMNtal の処理系, コンピュータソフトウェア, Vol. 25, No. 2(2008), pp. 47–77.

[13] 田口智大: 再帰的な文脈パターンマッチング機能を持つグラフ書換え言語の設計と効率的な実装手法, 卒業論文, 早稲田大学基幹理工学部情報理工学科, 2022.

[14] 奈良耕太: 再帰的な文脈パターンマッチング機能を持つグラフ書換え言語の設計と効率的な実装手法, 修士論文, 早稲田大学基幹理工学部情報理工学科, 2016.

[15] 奈良耕太, 上田和紀: パターン定義によるマッチングを導入したグラフ書換え言語とその実装, 日本ソフトウェア科学会第 *31* 回大会 *(2014* 年度*)* 講演論文集, 2014.

# A　各例題に対応する中間命令列

## A.1　評価文脈による計算順序の制御

```
Compiled Rule
  --atommatch:
  --memmatch:
    spec [1, 3]
    findatom [1, 0, 'eval'_1]
    subrule [2, 0, "context", [0, 1]]
    proceed []

Compiled Subrule @context
  spec [2, 5]
  getlink [2, 1, 0]
  branch [[
    subrule [3, 0, "context_0", [0, 1, 2]]
    branch [[
      subrule [4, 0, "context", [0, 1]]
      succreturn [4]
    ]]
    succreturn [3]
  ]]
  failreturn []

Compiled Subrule @context_0
  spec [3, 6]
  branch [[
    subrule [3, 0, "context_3", [0, 1, 2]]
    succreturn [3]
  ]]
  dereflink [3, 2, 2]
  func [3, 'add'_3]
  branch [[
    derefatom [4, 3, 0]
    isint [4]
    subrule [5, 0, "context_1", [0, 1, 3]]
    succreturn [5]
  ]]
  branch [[
    subrule [4, 0, "context_2", [0, 1, 3]]
    succreturn [4]
  ]]
  failreturn []

Compiled Subrule @context_1
  spec [3, 6]
  getlink [3, 2, 1]
  subrule [4, 0, "context_0", [0, 1, 3]]
  branch [[
    subrule [5, 0, "context_1", [0, 1, 2]]
    succreturn [5]
  ]]
  succreturn [4]

Compiled Subrule @context_2
  spec [3, 6]
  getlink [3, 2, 0]
  subrule [4, 0, "context_0", [0, 1, 3]]
  branch [[
    subrule [5, 0, "context_2", [0, 1, 2]]
    succreturn [5]
  ]]
  succreturn [4]

Compiled Subrule @context_3
  spec [3, 12]
  dereflink [3, 2, 2]
  func [3, 'add'_3]
  derefatom [4, 3, 0]
  isint [4]
  derefatom [5, 3, 1]
  isint [5]
  iadd [6, 4, 5]
  commit ["add", 0]
  removeatom [5, 0]
  removeatom [4, 0]
  removeatom [3, 0]
  copyatom [7, 0, 6]
  swaplink [7, 0, 3, 2]
  freeatom [3]
  freeatom [4]
  freeatom [5]
  allocset [9]
  succreturn [9]
```

## A.2　スキップリストの各駅の削除

```
Compiled Rule
  --atommatch:
  --memmatch:
    spec [1, 5]
    findatom [1, 0, 'listp'_2]
    getlink [2, 1, 0]
    getlink [3, 1, 1]
    subrule [4, 0, "skiplist, [0, 2, 3]]
    proceed []

Compiled Subrule @skiplist
  spec [3, 9]
  branch[[
    subrule [3, 0, "skiplist_0", [0, 1, 2]]
  ]]
  branch[[
    dereflink [3, 1, 4]
    func [3, 'a'_5]
    getlink [4, 3, 3]
    ispairedlink [2, 4]
    derefatom [5, 3, 2]
    isint [5]
    getlink [6, 3, 0]
    getlink [7, 3, 1]
    subrule [8, 0, "skiplist", [0, 6, 7]]
  ]]
  branch[[
    dereflink [3, 1, 4]
    func [3, 'b'_3]
    derefatom [4, 3, 1]
    isint [4]
    getlink [5, 3, 0]
    subrule [6, 0, "skiplist", [0, 5, 2]]
  ]]
  failreturn []

Compiled Subrule @skiplist_0
  spec [3, 3]
  branch [[
    subrule [3, 0, "skiplist_1", [0, 1, 2]]
  ]]
  failreturn []

Compiled Subrule @skiplist_1
  spec [3, 7]
  dereflink [3, 1, 4]
  func [3, 'a'_5]
  getlink [4, 3, 3]
  ispairedlink [2, 4]
  derefatom [5, 3, 2]
  isint [5]
```

```
  subrule [6, 0, "skiplist_2", [0, 3]]
  failreturn []

Compiled Subrule @skiplist_2
  spec [2, 5]
  getlink [2, 1, 0]
  subrule [3, 0, "skiplist_3", [0, 2]]
  subrule [4, 0, "skiplist_2", [0, 1]]
  failreturn []

Compiled Subrule @skiplist_3
  spec [2, 5]
  branch[[
    subrule [2, 0, "skiplist_5", [0, 1]]
    succreturn [2]
  ]]
  branch[[
    dereflink [2, 1, 2]
    func [2, 'b'_3]
    derefatom [3, 2, 1]
    isint [3]
    subrule [4, 0, "skiplist_4", [0, 2]]
    succreturn [4]
  ]]
  failreturn []

Compiled Subrule @skiplist_4
  spec [2, 5]
  getlink [2, 1, 0]
  subrule [3, 0, "skiplist_3", [0, 2]]
  subrule [4, 0, "skiplist_4", [0, 1]]
  failreturn []

Compiled Subrule @skiplist_5
  spec [2, 4]
  dereflink [2, 1, 2]
  func [2, 'b'_3]
  derefatom [3, 2, 1]
  isint [3]
  commit ["_list", 0]
  removeatom [2, 0]
  removeatom [3, 0]
  unify [2, 0, 2, 2, 0]
  freeatom [2]
  freeatom [3]
  succreturn []
```

### A.3 差分リストの連結命令列

```
Compiled Rule
  --atommatch:
  --memmatch:
    spec [1, 3]
    findatom [1, 0, 'list1'_1]
    subrule [2, 0, "dlist", [0, 1]]
    proceed []

Compiled Subrule @dlist
  spec [2, 5]
  getlink [2, 1, 0]
  subrule [3, 0, "dlist_0", [0, 2]]
  subrule [4, 0, "dlist", [0, 1]]
  failreturn []

Compiled Subrule @dlist_0
  spec [2, 5]
  branch[[
    subrule [2, 0, "dlist_2", [0, 1]]
    succreturn [2]
  ]]
  branch[[
    dereflink [2, 1, 2]
    func [2, '.'_3]
    derefatom [3, 2, 0]
    isint [3]
    subrule [4, 0, "dlist_1", [0, 2]]
    succreturn [4]
  ]]
  failreturn []

Compiled Subrule @dlist_1
  spec [2, 5]
  getlink [2, 1, 1]
  subrule [3, 0, "dlist_0", [0, 2]]
  subrule [4, 0, "dlist_1", [0, 1]]
  failreturn []

Compiled Subrule @dlist_2
  spec [2, 4]
  dereflink [2, 1, 0]
  func [2, '[]'_1]
  branch [[
    subrule [3, 0, "dlist_3", [0, 2]]
    succreturn [3]
  ]]
  failreturn []

Compiled Subrule @dlist_3
  spec [2, 5]
  findatom [2, 0, 'list2'_1]
  getlink [3, 2, 0]
  branch [[
    subrule [4, 0, "dlist_4", [0, 1, 2, 3]]
    succreturn [4]
  ]]
  failreturn []

Compiled Subrule @dlist_4
  spec [4, 8]
  branch[[
    subrule [4, 0, "dlist_5", [0, 1, 2, 3]]
    succreturn [4]
  ]]
  branch[[
    dereflink [4, 3, 2]
    func [4, '.'_3]
    derefatom [5, 4, 0]
    isint [5]
    getlink [6, 4, 1]
    subrule [7, 0, "dlist_4", [0, 1, 2, 6]]
    succreturn [7]
  ]]
  failreturn []

Compiled Subrule @dlist_5
  spec [4, 6]
  dereflink [4, 3, 0]
  func [4, '[]'_1]
  commit ["append", 0]
  removeatom [1, 0]
  removeatom [2, 0]
  unify [1, 0, 2, 0, 0]
  freeatom [1]
  freeatom [2]
  allocset [5]
  succreturn [5]
```

### A.4 差分リストの先頭要素削除命令列

```
Compiled Rule
  --atommatch:
  --memmatch:
    spec [1, 9]
    findatom [1, 0, 'list'_1]
    getlink  [2, 1, 0]
    dereflink [3, 2, 2]
    func [3, '.'_3]
    derefatom [4, 3, 0]
    isint  [4]
    getlink [5, 3, 1]
    subrule [6, 0, "dlist", [0, 5]]
    subrule [7, 0, "dlist_1", [0, 3, 4]]
    subrule  [8, 0, "dlist_2", [0, 1]]
    proceed []

Compiled Subrule @dlist
  spec [2, 6]
  branch [[
    subrule [2, 0, "dlist_0", [0, 1]]
    succreturn [2] ]]
  branch [[
    dereflink [2, 1, 2]
    func [2, '.'_3]
    derefatom [3, 2, 0]
    getlink [4, 2, 1]
    subrule [5, 0, "dlist", [0, 4]]
    succreturn [5] ]]
  failreturn []

Compiled Subrule @dlist_0
  spec [2, 4]
  dereflink [2, 1, 0]
  func [2, '[]'_1]
  allocset [3]
  succreturn [3]

Compiled Subrule @dlist_1
  spec [3, 3]
  commit ["rmvinit", 0]
  removeatom [1, 0]
  removeatom [2, 0]
  unify [1, 2, 1, 1, 0]
  freeatom [1]
  freeatom [2]
  succreturn []

Compiled Subrule @dlist_2
  spec  [2, 7]
  getlink  [2, 1, 0]
  dereflink [3, 2, 2]
  func [3, '.'_3]
  derefatom [4, 3, 0]
  isint [4]
  subrule [5, 0, "dlist_1", [0, 3, 4]]
  subrule [6, 0, "dlist_2", [0, 1]]
  failreturn []
```