

2017 年度 修士論文

Incremental Pattern Matching による  
階層グラフ書換え言語 LMNtal の高速化

提出日： 2018 年 1 月 26 日

指導： 上田 和紀 教授

早稲田大学院 基幹理工学研究科  
情報理工・情報通信専攻

学籍番号： 5116F082-3

松澤 望

## 概要

松澤 望

LMNtal は階層グラフ書換えに基づく言語モデルである。LMNtal はグラフのパターンマッチと書換えによって多様なシステムを表現する。グラフ書換え系におけるパターンマッチとは、書換え規則によって定義されたグラフのパターンについて、グラフの中からパターンに一致するサブグラフを探索することを指す。書換えでは、パターンマッチによって得られたサブグラフを別の構造のグラフに書き換える。これらのパターンマッチと書換えが繰り返し実行され、グラフが書き換わっていく。しかし、書換えによって変化した部分以外のグラフ構造についてパターンマッチによる探索を行っても探索結果は同じであり不要な計算である。そうした不要な再計算を防ぐためにパターンマッチ情報をキャッシュとして持つ Incremental Pattern Matching (IPM) がプロダクションシステムなどで使われている。IPM の実装の 1 つとして RETE と呼ばれるアルゴリズムがある。RETE はキャッシュ情報をネットワーク構造で管理することを特長として持つ。

本研究では、LMNtal のパターンマッチングの効率を改善することを目的に、プログラム変換による RETE の導入手法を提案した。提案したプログラム変換手法は、(1) ルールのパターンマッチ処理を複数のルールに分割することと、(2) 分割されたパターンマッチの結果をキャッシュするルールを生成することからなる。変換によって生成されたルールが、RETE ネットワークにおけるパターンマッチ結果のキャッシュの保持と更新を表現することによって効率化を図っている。本研究により、ルール実行による変化が全体グラフに比べて小さいプログラムに対して実行時間オーダーが  $O(N^3)$  から  $O(N)$  に改善されることが確認できた。さらにオーダーが改善されたことにより、問題のサイズを十分に大きくした場合の実行時間がプログラム変換前の時間より短くなることが確認できた。また、プログラム変換による実行時間のオーバーヘッドを削減するために LMNtal の中間命令によって RETE を表現する手法の効果について検討した。

# Abstract

Nozomi Matsuzawa

LMNtal is a language model based on hierarchical graph rewriting. LMNtal allows us to represent various systems by graph matching and rewriting. A pattern matching means to search subgraphs that match graph patterns defined by rewrite rules. In the rewriting, the subgraphs are rewritten to other graphs. The whole graph is dynamically transformed by repeating pattern matching and rewriting. However, the calculation of pattern matching for a graph which is not changed by rewriting is unnecessary because the result of pattern matching for the same graph is unchanged. In order to avoid recalculation, Incremental Pattern Matching (IPM) algorithms were developed and have been used in Production Systems. The RETE algorithm is one of the implementations of IPM. A characteristic of RETE is that the intermediate result of pattern matching is cached as a network structure.

In this thesis, we propose a method for introducing the RETE algorithm to LMNtal by program transformation. The proposed method is composed of two steps, (1) to split a process of pattern matching into a family of rules, and (2) to generate rules that create a cache for a result of the rule splitted. The execution of the translated rules corresponds to updating the cache in the RETE network. As the result of this study, we improved the computational complexity of programs (in which the changes due to rule execution are smaller than the whole graph) from  $O(N^3)$  to  $O(N)$ . We confirmed that, the execution time was shorter than the execution times of program before transformation when the problem size was large. In addition, in order to reduce the overhead of execution time due to the program transformation, we discussed a method that represents RETE by intermediate instructions of LMNtal.

# 目次

第 1 章	はじめに	1
1.1	研究の背景と目的 . . . . .	1
1.2	論文の構成 . . . . .	2
第 2 章	LMNtal	3
2.1	背景 . . . . .	3
2.2	構成要素の概要 . . . . .	3
2.3	基本構文 . . . . .	4
2.4	操作的意味論 . . . . .	6
2.5	処理系の概要 . . . . .	7
2.6	HyperLMNtal . . . . .	10
第 3 章	Incremental Pattern Matching	12
3.1	プロダクションシステム . . . . .	12
3.2	RETE アルゴリズム . . . . .	13
3.3	ネットワークの共有 . . . . .	17
3.4	繰り返し発火されることの制御 . . . . .	17
3.5	関連研究 . . . . .	17
第 4 章	プログラム変換	19
4.1	RETE アルゴリズムの導入 . . . . .	19
4.2	プログラム変換の概要 . . . . .	19
4.3	RETE ネットワークの LMNtal における表現 . . . . .	20
4.4	ルールの生成 . . . . .	21
4.5	RETE ルールの制御フロー . . . . .	23

目次	ii
4.6	RETE による uniq 制約の表現 . . . . . 25
第 5 章	評価実験 26
5.1	RAM マシンシミュレータに関する計測 . . . . . 26
5.2	最短経路問題に関する計測 . . . . . 29
5.3	バブルソートに関する計測 . . . . . 30
第 6 章	まとめと今後の課題 33
6.1	まとめ . . . . . 33
6.2	今後の課題 . . . . . 33
謝辞	36
参考文献	37
外部発表	39
A.1	RAM マシンシミュレータ . . . . . 40
A.2	最短経路問題 . . . . . 41
A.3	バブルソート . . . . . 44

# 目次

2.1	階層グラフ . . . . .	3
2.2	LMNtal の基本構文 . . . . .	5
2.3	LMNtal プロセスの構造合同 . . . . .	6
2.4	LMNtal プロセスの遷移関係 . . . . .	7
2.5	LMNtal 処理系ツールチェーン . . . . .	7
2.6	$a(X), b(X) :- a(X), c(X)$ の中間命令列 . . . . .	8
2.7	各種中間命令の詳細 . . . . .	9
2.8	アトムリスト . . . . .	11
2.9	HyperLMNtal のプログラム例 . . . . .	11
2.10	HyperLMNtal のグラフ構造 . . . . .	11
3.1	プロダクションシステムの例 . . . . .	13
3.2	IPM の概要 [1] . . . . .	14
3.3	RETE ネットワークの例 [2] . . . . .	15
3.4	Join ノードの働き . . . . .	16
3.5	ネットワークの共有 . . . . .	17
4.1	RETE ルールの制御フロー . . . . .	23
4.2	ルールの逐次実行の例 . . . . .	24
4.3	ルール停止条件の例 . . . . .	24
4.4	RETE ルールを制御するルール . . . . .	25
5.1	RAM マシンシミュレータの上で実行される命令列 . . . . .	27
5.2	RAM マシンシミュレータの LMNtal プログラム . . . . .	28
5.3	RAM マシンシミュレータの評価結果 . . . . .	29

---

5.4	最短経路問題の入力 . . . . .	30
5.5	最短経路問題の LMNtal プログラム . . . . .	30
5.6	最短経路問題の評価結果 . . . . .	31
5.7	最短経路問題の中間命令数比較 . . . . .	31
5.8	バブルソートの LMNtal プログラム . . . . .	32
5.9	バブルソートの評価結果 . . . . .	32
6.1	HyperLMNtal のプログラム例 . . . . .	35

# 表目次

5.1	実験環境 . . . . .	26
-----	----------------	----



# 第 1 章

## はじめに

### 1.1 研究の背景と目的

階層グラフ書換え言語 LMNtal [3] は、階層グラフを対象にグラフのパターンマッチと書換えの繰り返しによって多様なシステムをモデル化するモデル記述言語である。階層グラフ構造は社会組織から生体に至るまであらゆる場面で見られる構造であり汎用的なモデルを記述することに適している。例えば SNS におけるユーザ間の関係の変化や、ネットワーク上のホスト間のパケットのやり取りなど様々なものが階層グラフ構造の書き換えによって表現することができる。しかし、ユーザが他のユーザをフォローしたり、あるホストからパケットを送信するなどの一度の動作によって変化するグラフ構造は、そのモデルのグラフ全体のサイズに比べて非常に小さいことがほとんどである。そのような大規模なグラフの上で書き換えを行うとき、1つ1つの書き換えの度にグラフ全体の中から書き換え対象のパターンマッチを網羅的行うと、変化のないグラフ構造に対するパターンマッチの再計算が頻繁に発生し非効率的である。そうした不要な再計算を防ぐためにパターンマッチ情報をキャッシュとして持つ Incremental Pattern Matching (IPM) がプロダクションシステム [1] などで使われている。

本研究の目的は、LMNtal のマッチングにかかる計算コストを削減することで大規模なグラフを扱えるようにすることである。本研究では、LMNtal のパターンマッチングの効率を改善することを目的に、プログラム変換によって IPM の実装の 1 つである RETE を LMNtal へ導入する手法を提案した。提案したプログラム変換手法は、(1) ルールのパターンマッチ処理を複数のルールに分割することと、(2) 分割されたパターンマッチの結果をキャッシュするルールを生成することからなる。変換によって生成されたルールが、RETE ネットワークにおけるパターンマッチ結果のキャッシュの保持と更新を表現する

ことによって効率化を図っている。本研究により、ルール実行による変化が全体グラフに比べて小さいプログラムに対して実行時間オーダーが  $O(N^3)$  から  $O(N)$  に改善されることが確認できた。さらにオーダーが改善されたことにより、問題のサイズを十分に大きくした場合の実行時間がプログラム変換前の時間より短くなることが確認できた。

## 1.2 論文の構成

本論文の構成は以下のとおりである。第2章では、階層グラフ書換え言語 LMNtal の定義とその処理系 SLIM について解説する。第3章ではプロダクションシステムを基に IPM の実装のひとつである RETE について解説する。第4章では RETE を LMNtal へ導入するためのプログラム変換手法について述べる。第5章では RETE による効率化についての実験結果を示す。第6章ではまとめと今後の課題について述べる。

## 第 2 章

# LMNtal

この章では階層グラフ書換え言語 LMNtal[3] の基本概念とその実行時処理系 SLIM[4] について述べる。

### 2.1 背景

LMNtal は階層グラフを対象にグラフのパターンマッチと書換えの繰り返しによって多様な計算システムをモデル化するモデル記述言語である。簡潔な計算モデルであると同時に実用的なプログラミング言語であることを目指している。

### 2.2 構成要素の概要

LMNtal は一級データとして階層グラフを持ち、その階層グラフを書き換えるルールを定義することでプログラムを構成する。階層グラフは図 2.1 に示すような構造を持つ。グラフの頂点をアトムと呼び、アトム同士を接続するエッジをリンクと呼ぶ。グラフは膜によってグループ分けされ、膜の入れ子によって階層構造を表す。

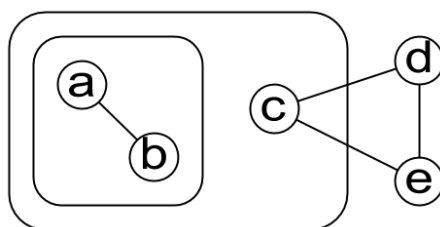


図 2.1 階層グラフ

グラフを書き換える規則はルールと呼ばれ、次のような形式で記述される。

$$Head :- Body$$

*Head* と *Body* にはグラフ構造のパターンが記述され、*Head* に一致するグラフが見つかったときそのグラフが *Body* に置き換えられる。ルールは膜で囲われた階層ごとに 0 個以上存在して、ルールの適用範囲はその階層に存在するグラフに制限される。

## 2.3 基本構文

LMNtal の構文を図 2.2 に示す。  $p$  は名前を表し小文字から始まる識別子で表記する。  $X_i$  はリンクを表し大文字から始まる識別子で表記する。構文は主にプロセス  $P$  とプロセステンプレート  $T$  からなる。直感的には  $P$  はデータとして現れるグラフを表し、 $T$  はルールの上で現れるグラフパターンを表す。

### 2.3.1 プロセス

アトムは引数に  $m \geq 0$  本のリンクを持ち、同じ識別子  $X_i$  のリンク同士は接続関係にある。また一般的なグラフとは異なり、アトムの引数には順序があり  $p(X_1, X_2)$  と  $p(X_2, X_1)$  は区別される。アトムの名前  $p$  と引数の数 (価数) のペアをファンクタと呼ぶ。LMNtal のアトムは多重集合であり、同一のファンクタを持つアトムが複数存在することを許す。また、予約アトム名として  $=$  があり  $=(X, Y)$  は  $X$  と  $Y$  を接続することを表す。

プロセス  $P$  に 1 回だけ出現するリンクを  $P$  の自由リンクと呼び、 $P$  にちょうど 2 回出現するリンクを  $P$  の局所リンクと呼ぶ。また、プロセスには「同じリンク名が 2 回を超えて出現してはならない」というリンク条件がある。

### 2.3.2 プロセステンプレート

ルールは膜に所属して、その膜や子孫膜の内容を書き換えることができる。プロセス文脈とルール文脈はプロセステンプレートの中に現れ、不特定のアトムやルールに対するマッチングを可能にする。プロセス文脈は膜の中のプロセス全体にマッチして、ルール文脈は膜の中の全てのルールの集合にマッチする。プロセス文脈  $\$p[X_1, \dots, X_m \mid A]$  にマッチしたプロセスは自由リンクとして  $X_1, \dots, X_m$  を持たなければならない。また、 $A$  が  $*X$  のときは  $X_1, \dots, X_m$  以外に 0 本以上の自由リンクを持つことを表す。

### 2.3.3 拡張構文

LMNtal では数値や文字列などを  $x(1)$  や  $y(\text{"text"})$  のようにアトムの引数として表現する. あるアトムの引数がどの型を持つか検査するために LMNtal では型付きプロセス文脈とガードと呼ばれる拡張を導入している.

$$\textit{Head} \text{ :- } \textit{Guard} \mid \textit{Body}$$

ガードを持つ例を用いて説明する.

$$a(X), \$n[X] \text{ :- } \textit{int}(\$n), \$n > 0, \$m = \$n - 1 \mid a(X), \$m[X]$$

このルールはアトム  $a$  の引数  $\$n$  が整数でありかつ 0 以上であるときのみ,  $\$n$  から 1 を引いた数をアトム  $a$  に接続することを表す.  $\$n[X]$  を型付きプロセス文脈と呼び,  $X$  に繋がった閉じたプロセス全体にマッチする. ガードには左辺でマッチした型付きプロセス文脈の型の判定や比較演算・算術演算を記述することができる.

---

$P$	$::=$	0	(空)
		$p(X_1, \dots, X_m)$ ( $m \geq 0$ )	(アトム)
		$P, P$	(分子)
		$\{P\}$	(セル)
		$T \text{ :- } T$	(ルール)
$T$	$::=$	0	(空)
		$p(X_1, \dots, X_m)$ ( $m \geq 0$ )	(アトム)
		$T, T$	(分子)
		$\{T\}$	(セル)
		$T \text{ :- } T$	(ルール)
		$@p$	(ルール文脈)
		$\$p[X_1, \dots, X_m \mid A]$ ( $m \geq 0$ )	(プロセス文脈)
		$p(*X_1, \dots, *X_m)$ ( $m > 0$ )	(アトム集団)
$A$	$::=$	$[]$	(空)
		$*X$	(リンク束)

---

図 2.2 LMNtal の基本構文

## 2.4 操作的意味論

本節では、LMNtal の操作的意味論をプロセスの構造合同関係と遷移関係の定義によって示す。

### 2.4.1 構造合同

プロセス同士の構造合同関係を図 2.3 に示す。これらの関係にあるプロセスは本質的に等価なプロセスであるとみなし、互いに変換可能である。(E1) から (E3) は分子が多重集合であることを示す。(E4) は局所リンクの名前を任意に付け替えられる  $\alpha$  変換を許すものである。(E5) から (E6) は  $\equiv$  を合同関係にするための構造規則である。(E7) から (E9) は  $=$  アトムに関する関係で、(E7) は自己完結したループが 0 と等価であることを示し、(E8) は  $=$  の対称性、(E9) は  $=$  アトムが他のプロセスに吸収される、あるいは他のプロセスから放出されることを示す。

---

(E1)	$0, P$	$\equiv$	$P$	
(E2)	$P, Q$	$\equiv$	$Q, P$	
(E3)	$P, (Q, R)$	$\equiv$	$(P, Q), R$	
(E4)	$P$	$\equiv$	$P[Y/X]$	( $X$ は $P$ の局所リンク名)
(E5)	$P \equiv P'$	$\Rightarrow$	$P, Q \equiv P', Q$	
(E6)	$P \equiv P'$	$\Rightarrow$	$\{P\} \equiv \{P'\}$	
(E7)	$X = X$	$\equiv$	$0$	
(E8)	$X = Y$	$\equiv$	$Y = X$	
(E9)	$X = Y, P$	$\equiv$	$P[Y/X]$	(ただし $P$ はアトムで $X$ は $P$ の自由リンク名)

---

図 2.3 LMNtal プロセスの構造合同

### 2.4.2 遷移関係

LMNtal の計算とはルールの適用によりプロセスが遷移することである。プロセス間の遷移関係を図 2.4 に示す。

(R1) は書換え条件の局所性を表し、LMNtal の並行性を表す。(R2) はセル内の計算を外部と独立に進めることができることを表す。(R3) は構造合同を遷移関係に取り込むための規則である。(R4) と (R5) は  $=$  アトムの膜を超える移動規則を表す。(R6) は

LMNtal の中心となるルールの適用に関する規則であり，同じ膜に共存するプロセスとルールとの反応を表している．

$$\frac{P \rightarrow P'}{P, Q \rightarrow P', Q} \text{ (R1)}$$

$$\frac{P \rightarrow P'}{\{P\} \rightarrow \{P'\}} \text{ (R2)}$$

$$\frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'} \text{ (R3)}$$

(R4)  $\{X = Y, P\} \rightarrow X = Y, \{P\}$  (ただし  $X$  と  $Y$  は  $\{X = Y, P\}$  の自由リンク名)

(R5)  $X = Y, \{P\} \rightarrow \{X = Y, P\}$  (ただし  $X$  と  $Y$  は  $P$  の自由リンク名)

(R6)  $T\theta, (T :- U) \rightarrow U\theta, (T :- U)$

図 2.4 LMNtal プロセスの遷移関係

## 2.5 処理系の概要

LMNtal 処理系のツールチェーンの概要図を図 2.5 に示す．LMNtal プログラムはコンパイラによって中間命令列に変換され，その中間命令列が実行時処理系 SLIM によって解釈・実行される．SLIM は約 36000LOC の C コードから成り，LMNtal プログラムの実行の他にプログラムのモデルチェックを行うツールが実装されている．

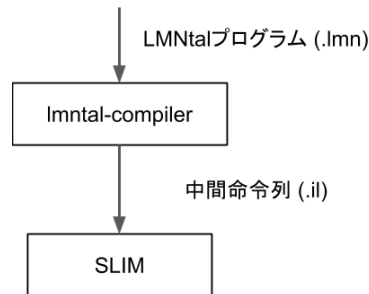


図 2.5 LMNtal 処理系ツールチェーン

### 2.5.1 LMNtal の中間命令

LMNtal のルール  $a(X), b(X) :- a(X), c(X)$  をコンパイルしたときに出力される中間命令列の例を図 2.6 に示す. `commit` 命令より上の命令列がルールの左辺に対応し, `commit` 命令より下の命令列が右辺に対応する. それぞれの中間命令の詳細について図 2.7 に示す.

1	Compiled Rule
2	--atommatch:
3	spec [2, 5]
4	--memmatch:
5	spec [1, 5]
6	findatom [1, 0, 'a'_1]
7	deref [2, 1, 0, 0]
8	func [2, 'b'_1]
9	commit ["_aXbX", 0]
10	removeatom [1, 0, 'a'_1]
11	removeatom [2, 0, 'b'_1]
12	newatom [3, 0, 'a'_1]
13	newatom [4, 0, 'c'_1]
14	newlink [3, 0, 4, 0, 0]
15	freeatom [1]
16	freeatom [2]
17	proceed []

図 2.6  $a(X), b(X) :- a(X), c(X)$  の中間命令列

### 2.5.2 処理系 SLIM の実行の流れ

SLIM は逐次計算機として設計されており, 中間命令列を逐次的に処理する. 図 2.6 を例に SLIM の処理の流れを見る. まず, SLIM は中間命令列の上からコンパイルされた 1 つのルールを見つける. 見つけたルールのヘッドの命令列を上から読み込み--memmatch



---

`findatom` [*dstatom*, *srcmem*, *funcref*]

膜 *srcmem* の内部にファンクタが *funcref* であるアトムがあったとき命令に成功して、そのアトムへの参照を *dstatom* に代入する。

`deref` [*dstatom*, *srcatom*, *srcpos*, *dstpos*]

アトム *srcatom* の第 *srcpos* 引数のリンクが、あるアトムの第 *dstpos* 引数に接続していたら命令成功して、そのアトムへの参照を *dstatom* に代入する。

`func` [*srcatom*, *funcref*]

アトム *srcatom* のファンクタが *funcref* であるとき命令に成功する。

`commit` [*rulename*, *lineno*]

ルールのマッチングに成功したときに実行される命令。 *rulename*, *lineno* はデバッグのための情報を持つ。

`removeatom` [*srcatom*, *srcmem*, *funcref*]

ファンクタが *funcref* であるアトム *srcatom* を膜 *srcmem* から削除する。

`newatom` [*dstatom*, *srcmem*, *funcref*]

ファンクタが *funcref* であるアトムを膜 *srcmem* に生成して、そのアトムへの参照を *dstatom* に代入する。

`newlink` [*atom1*, *pos1*, *atom2*, *pos2*, *mem*]

膜 *mem* にあるアトム *atom1* の第 *pos1* 引数とアトム *atom2* の第 *pos2* 引数の間に新しくリンクを生成する。

からパターンマッチ処理が始まる。

1. `findatom [1, 0, 'a'_1]` は 1 価の名前 `a` であるファンクタを持つアトムを探索してレジスタ 1 に代入する。
2. `deref [2, 1, 0, 0]` は取得したアトムの第 0 引数の接続先があるアトムの第 0 引数であるときレジスタ 2 にそのアトムの参照を代入する
3. `func [2, 'b'_1]` はレジスタ 2 のアトムのファンクタが 1 価の `b` であることを安定する

このとき `deref` 命令と `func` 命令は失敗することがある。これらの命令に失敗すると SLIM は直前の `findatom` 命令にバックトラックする。バックトラックしたあとの `findatom` はそのとき探索していたアトムとは別のアトムを探索し、以降の命令を再び処理していく。こうして命令の失敗とバックトラックを繰り返して、`commit` 命令に到達したときそのルールのパターンマッチングに成功する。

### 2.5.3 アトムの管理

`findatom` 命令はあるファンクタのアトムを全体グラフから探索する。SLIM では探索を効率よく行うため、アトムの集合をファンクタがキーでそのファンクタを持つアトムの List を値として持つ HashMap によって管理している。この構造をアトムリスト (図 2.8) と呼ぶ。この構造によってあるファンクタを探索する処理は  $O(1)$  で行われ、またアトムの生成と削除はアトムリストへの要素の挿入と削除によって実現される。

## 2.6 HyperLMNtal

HyperLMNtal [5] とは LMNtal の拡張言語モデルである。HyperLMNtal は LMNtal におけるリンクの他に HyperLink と呼ばれる要素が追加され、数学における Hypergraph を表現することができる。プログラムの例とその構造を図 2.9 と図 2.10 に示す。通常のリンクの前に '!' をつけることで HyperLink を表現する。通常のリンクと異なりリンクの出現条件の制約がなくプロセス内に同じ HyperLink 名が 0 個以上現れてもよい。また、ガード条件 `num` を使うことで HyperLink に繋がっているアトムの数を得ることができる。

SLIM には HyperLink を用いた効率的なパターンマッチング手法 [6] が存在する。

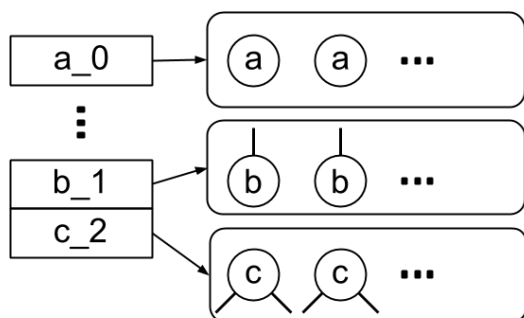


図 2.8 アトムリスト

```

1 init.
2 init :- a(!H0), b(!H0), c(!H0), d(!H1).
3
4 a(!H) :- num(!H) > 0 | e(!H).

```

図 2.9 HyperLMNtal のプログラム例

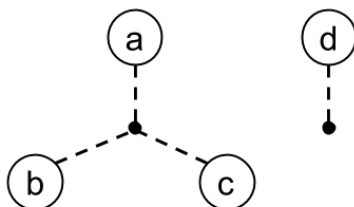


図 2.10 HyperLMNtal のグラフ構造

## 第 3 章

# Incremental Pattern Matching

本章では Incremental Pattern Matching (IPM) に基づくパターンマッチを採用しているプロダクションシステムを解説し、プロダクションシステムを例に IPM の実装の 1 つである RETE について解説する。この章は文献 [1] と文献 [2] を参考に記述した。

### 3.1 プロダクションシステム

プロダクションシステム [1] はルールベースのプログラミング言語で、ロボットの知的行動モデルの記述や複雑な問題解決のためのエキスパートシステムの記述などに用いられてきた。プロダクションシステムはルールを格納する Production Memory (PM) と、データを格納する Working Memory (WM) によって構成される。ルールは条件部 (LHS) と動作部 (RHS) からなり、条件部を満たすデータが WM 内に存在するとき動作部が実行される。動作部は WM へのデータの追加・削除・更新などの動作列で構成される。プロダクションシステムの実装の 1 つである OPS5[7] のプログラム例を図 3.1 に示す。

WM のデータは (`identifier [^attribute value]*`) という形式で表され、属性と値の組を並べて記述する。PM のルールは (`p rule-name LHS --> RHS`) で表される。LHS を構成するデータのパターンを条件要素と呼ぶ。条件要素に現れる `<x>` はパターン変数と呼ばれ、その条件要素がデータにマッチしたとき変数に具体値が束縛される。また、条件要素間の同じ変数名は同じ具体値が束縛されることを明示している。つまり図 3.1 のルール `rule1` は「color 属性が red であるデータ `x` と color 属性が green であるデータ `y` が存在して、`x` の `on` 属性の値が `y` である」とき条件部が全て満たされる。RHS は LHS にマッチした要素への操作が記述され、`make`, `remove`, `modify` の 3 つの命令がありそれぞれデータの生成・削除・修正を行う。

プロダクションシステムの実行サイクルは、条件照合、競合解決、実行の繰り返しである。条件照合フェーズは全てのルールの特徴要素とその時点における WM のデータとのマッチングを行い、ルールとそのルールにマッチングするデータの組（インスタンスエーション）を生成する。競合解決フェーズでは特定の戦略を基にインスタンスエーションの集合から1つ選び出す。実行フェーズでは選択されたインスタンスエーションのルールを実行する。

```
1 # Working Memory Element
2 (box1 ^color red)
3 (box2 ^color green)
4 (box1 ^on box2)
5 (box2 ^on table)
6
7 # Production Rule
8 (p rule1
9   (<x> ^color red)
10  (<y> ^color green)
11  (<x> ^on <y>)
12  -->
13  (modify 1 ^color blue)
14 )
```

図 3.1 プロダクションシステムの例

## 3.2 RETE アルゴリズム

OPS5 では条件照合アルゴリズムとして RETE と呼ばれるアルゴリズムを採用している。RETE アルゴリズム [8] は IPM の実装のひとつで、OPS5 をはじめ CLips[9] や Jess[10] などのプロダクションシステムに導入されている。RETE の詳細なアルゴリズムについては文献 [2] が詳しい。

RETE では、実行サイクルの条件照合の度に全てのインスタンスエーションを計算するのではなく、前回の条件照合のデータをキャッシュとして保持して、実行によって変化したデータからインスタンスエーションを更新する。RETE による IPM の概要を図 3.2

に示す．本論文ではパターンの差分については考慮せず，データの差分についてのみ議論する．

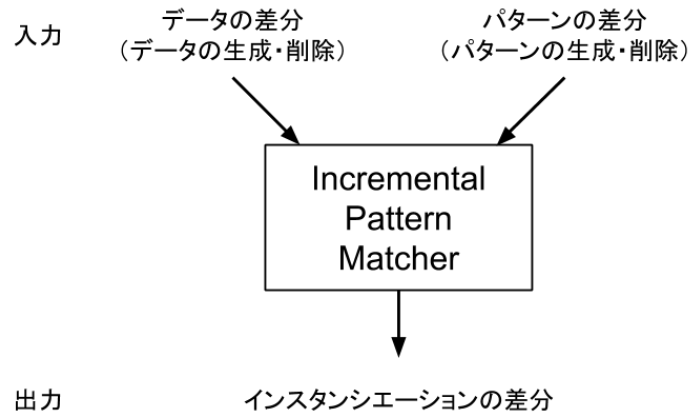


図 3.2 IPM の概要 [1]

RETE はデータの差分から効率的にマッチング可能データを計算するために，パターンの部分的なマッチング情報をキャッシュとして保持する．これらのキャッシュは RETE ネットワークと呼ばれる構造で管理される．RETE ネットワークの例を図 3.3 に示す．RETE ネットワークは条件要素内のテストを行う  $\alpha$  ネットワークと，条件要素間のテストを行う  $\beta$  ネットワークから構成される．

実行フェーズで WM にデータが追加されたとき，そのデータがトークンとして RETE ネットワークに流し込まれる．トークンはネットワーク上部から条件要素内テストと条件要素間テストを経てネットワーク内のメモリを更新する．

### 3.2.1 $\alpha$ ネットワーク

$\alpha$  ネットワークではトークンを受け取り，そのトークンを適切な  $\alpha$  メモリに分類する． $\alpha$  メモリはルールの条件要素と対応していて，条件要素を満たすトークンが格納される．トークンの分類は条件要素内テストノード（テストノード）で行われる．まず初めにテストノードでトークンを属性で分類する．`attr=on?`と書かれたノードは属性の名前が `on` であるトークンを後ろに流す．次に属性の値で分類を行う．`^color=red?`と書かれたノードは `color` 属性の値が `red` であるトークンを後ろに流す．振り分けられたトークンは  $\alpha$  メモリに到達すると，メモリに格納されるとともに後続の  $\beta$  ネットワークに同じトークンを流す．

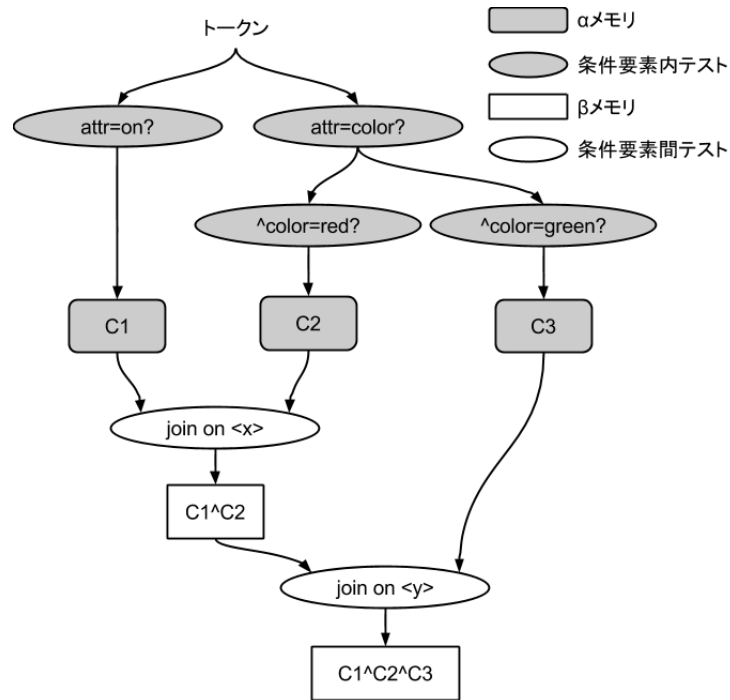


図 3.3 RETE ネットワークの例 [2]

### 3.2.2 β ネットワーク

β ネットワークではルールの条件要素間の変数束縛の一致をテストする．条件要素間テストを行うノードは Join ノードと呼ばれる．Join ノードの入力側は2つのメモリと接続されており，接続されたメモリ内の条件要素間のテストを行う．Join ノードのテストは入力側のどちらかにトークンが到着したときに行われる．WM が以下のような状態であるときのトークンが到着したときの Join ノードの挙動を図 3.4 に示す．

W1: (box1 ^on box2)  
 W2: (box2 ^on table)  
 W3: (box1 ^color red)

join on <x>と書かれた Join ノードは左の条件要素と右の条件要素で変数 <x> が同じ値を束縛していることをテストする．ここでは条件要素 C1 と C2 にマッチしたデータの <x> が一致することをテストする．

C1: ( $\langle x \rangle \wedge_{\text{on}} \langle y \rangle$ )

C2: ( $\langle x \rangle \wedge_{\text{color red}}$ )

W3 トークンが Join ノードに到達したとき, Join の反対側のメモリ全てに対して  $\langle x \rangle$  の一致をテストして, テストに成功した組み合わせのみ  $\beta$  メモリに格納される. 図では W1 と W3 の組み合わせのみテストに成功した.  $\beta$  メモリに格納された組み合わせは再びトークンとして後ろのネットワークへ流される.

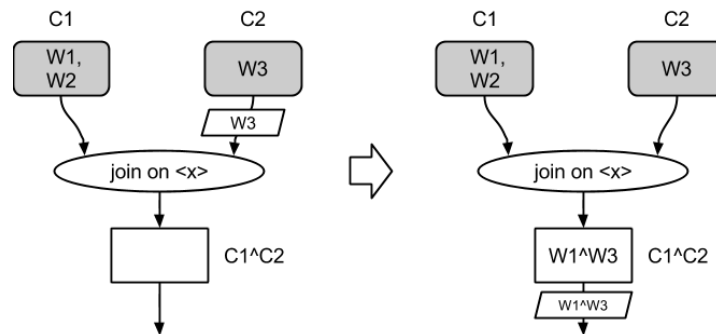


図 3.4 Join ノードの働き

### 3.2.3 データの削除

実行フェーズでは WM のデータの削除も行われる. WM のデータが削除されたとき, RETE ネットワークも該当するメモリのエントリを削除する必要がある. RETE ではメモリの削除のために負のトークンという概念を用いている. 負のトークンは正のトークンと同じように RETE ネットワークに流し込まれる.  $\alpha \cdot \beta$  におけるテストも同じように行われるが, メモリに負のトークンが到着したときそのメモリから負のトークンと一致するエントリを削除する.

しかし, 文献 [2] でも指摘されているようにこの方法はデータの追加と同じだけコストがかかりあまり効率が良くない. そこで文献 [2] では, WM 上のデータやメモリとトークンに対するポインタをリストか木構造で管理し, 削除する必要がある要素をポインタを辿って削除する手法を提案している.



### 3.3 ネットワークの共有

異なるルールにおいて条件要素が共通する場合にネットワークノードの共有によって効率化できる。ネットワークの共有の例を図 3.5 に示す。この例では2つのルールがネットワークを共有しており、条件要素 C1 と C2 の組み合わせが共有されている。ネットワークノードが共有されることにより、条件要素内テストや条件要素間テストの回数が減り、またメモリも削減される。

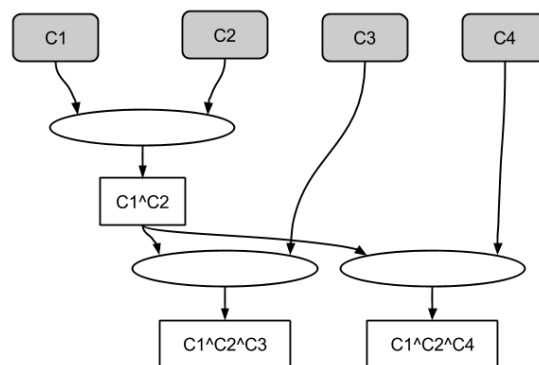


図 3.5 ネットワークの共有

### 3.4 繰り返し発火されることの制御

プロダクションシステムでは、ルールの適用に選ばれたインスタンスエーションが再度ルール適用として選ばれて繰り返し発火されることを防ぐために、一度選択されたインスタンスエーションを競合集合から取り除かれる。

## 3.5 関連研究

### 3.5.1 RETE の発展

RETE はマッチングアルゴリズムとしてスタンダードなものであるが、RETE のさらに発展として TREAT[11] や LEAPS[12] などのアルゴリズムがある。

TREAT は  $\beta$  メモリを持たないことを特徴として持つ。RETE では条件部の部分的なマッチングのキャッシュを  $\beta$  メモリに保持していたが、ルールの実行によるデータの変化

がこれらの  $\beta$  メモリの殆どを更新するような場合があると逆に性能が悪化することが考えられる．そこで，TREAT では  $\beta$  メモリをなくして実行サイクルごとに全ての  $\beta$  メモリへの Join 演算を計算するようにしている．

### 3.5.2 Constraint Handling Rules

Constraint Handling Rules (CHR) [13] は制約プログラミング言語である．また，CHR の実装の 1 つである JCHR [14] は LEAPS と近いパターンマッチアルゴリズムを採用しており，RETE と LEAPS との比較が議論 [15] されている．

CHR は制約の集合を書換え規則によって書換えていく．CHR の基本文法は次の通りである．

$$\begin{array}{ll}
 \text{Simplification rule} & h_1, \dots, h_n \iff g_1, \dots, g_m \mid b_1, \dots, b_k \\
 \text{Propagation rule} & h_1, \dots, h_n \implies g_1, \dots, g_m \mid b_1, \dots, b_k \\
 \text{Simpagation rule} & h_1, \dots, h_l \setminus h_{l+1}, \dots, h_n \iff g_1, \dots, g_m \mid b_1, \dots, b_k
 \end{array}$$

$h_i$  と  $b_i$  は CHR におけるひとつの制約を表し， $g_1, \dots, g_m$  はルールの適用を制御するガード条件である．ルールは Simplification ルールと Propagation ルールと Simpagation ルールの 3 種類ある．

Simplification ルールは制約の集合の中から  $h_1, \dots, h_n$  に一致する制約を削除して， $b_1, \dots, b_k$  を制約集合に追加する．Propagation ルールは制約の集合の中から  $h_1, \dots, h_n$  に一致する制約があったとき，それらの制約を削除せずに  $b_1, \dots, b_k$  を制約集合に追加する．このとき Propagation ルールは  $h_1, \dots, h_n$  に一致したことがある制約を履歴として管理して，同じ制約に 2 回以上適用されないように制限されている．Simpagation ルールは Simplification ルールと Propagation ルールの複合版で， $h_1, \dots, h_l$  と  $h_{l+1}, \dots, h_n$  に一致する制約があったとき， $h_1, \dots, h_l$  は削除して  $h_{l+1}, \dots, h_n$  は削除せずに残す．

## 第 4 章

# プログラム変換

この章では，LMNtal のパターンマッチングに RETE アルゴリズムを導入するための手法について述べる．

### 4.1 RETE アルゴリズムの導入

プロダクションシステムのルールは LMNtal のルールとほぼ 1 対 1 の対応がとれる．条件要素は LMNtal ルール左辺のアトムにあたり，条件要素間の変数束縛のテストは LMNtal のリンクの接続関係のテストと見ることができる．このことから，プロダクションシステムで用いられている RETE によるパターンマッチ手法を LMNtal に応用することが考えられる．

本研究では前節の問題を解決するために LMNtal のマッチングアルゴリズムに RETE を導入することを提案した．ここで，LMNtal 処理系 SLIM [4] を拡張するのではなく与えられた LMNtal ルールを RETE と同じようなパターンマッチングを行うルールへ変換する手法をとった．これは処理系を拡張するより作業コストが小さく，また実行効率を調査するには十分であると考えたためである．

### 4.2 プログラム変換の概要

変換手法の主な考えは (1) ひとつのルールをいくつかのサブルールに分割し，データの差分に関連のあるサブルールだけ実行されるようにすることと，(2) 部分的なマッチングデータをキャッシュとして生成するルールを生成することである．これらのルールの実行が RETE ネットワークによるキャッシュの保持と更新を表していて，効率的なマッチン

グを実現している．説明のため LMNtal ルールを以下のように表現する． $h_i, b_j$  は1つのアトムを表している．

$$h_1, \dots, h_n :- b_1, \dots, b_m$$

このルールからプログラム変換によって生成される各ルールについて解説する．

変換元ルールは主に  $\alpha$  ルール， $\beta$  ルール，書き換えルールの3つのサブルールに分割される． $\alpha$  ルールは変換元ルールの左辺を構成する要素  $h_i$  ごとに生成される．

$$\begin{aligned} h_1 & :- \langle h_1 \rangle, token_{h_1}. \\ \dots & \\ h_n & :- \langle h_n \rangle, token_{h_n}. \end{aligned}$$

$\langle h_i \rangle$  は  $h_i$  にマッチしたことを示すキャッシュを表し， $token_{h_i}$  は  $h_i$  が新規に生成されたデータであることを表すタグである． $\langle h_i \rangle$  と  $token_{h_i}$  は実際には LMNtal のデータ構造であるがここでは簡単のため抽象表現を用いる．

$\beta$  ルールは部分的なマッチングのキャッシュを生成する役割を持つ．これらのルールは  $token_{h_i}$  があるときのみ反応するため，変化したデータに関連するルールのみ実行されるような制御を実現している．

$$\begin{aligned} token_{h_1}, \langle h_2 \rangle & :- \langle h_1, h_1 \rangle, token_{h_1, h_2}. \\ token_{h_1, h_2}, \langle h_3 \rangle & :- \langle h_1, h_2, h_3 \rangle, token_{h_1, h_2, h_3}. \\ \dots & \\ token_{h_1, \dots, h_{n-1}}, \langle h_n \rangle & :- \langle h_1, \dots, h_n \rangle, token_{h_1, \dots, h_n}. \end{aligned}$$

書き換えルールは変換元ルールの左辺をキャッシュに置き換えたルールを生成する．

$$\langle h_1, \dots, h_n \rangle :- b_1, \dots, b_m.$$

以上のルール変換をプログラムの全てのルールに対して適用する．

### 4.3 RETE ネットワークの LMNtal における表現

前節で用いたトークンとキャッシュという抽象表現を LMNtal では HyperLink によって表現した．例えば  $a(X)$  というアトムがメモリに格納されていることを

`mem_a(!Ref), a(X, !Ref)`

と表し，同様に  $a(X)$  というアトムのトークンを

`token_a(!Ref), a(X, !Ref)`

と表した．つまりアトム最後の引数に `HypreLink` を付加して，それをメモリやトークンを表すアトムから参照するようにしている．

## 4.4 ルールの生成

各ルールについてより詳細な説明をしていく．

### 4.4.1 $\alpha$ ルール

$\alpha$  ルールは RETE の  $\alpha$  ネットワークの処理を行うルールである．

$$h_i(\dots, L_j, \dots) :- h_i\_rete(\dots, L_i, \dots, !Ref), token\_ \langle h_i \rangle (!Ref)$$

これはあるルールの実行によって生成されたアトム  $h_i$  を検出して，トークンを生成するルールである．変換元のプログラムに出現するアトム名と衝突を避けるために，右辺では  $h_i$  の名前に `rete` という文字を追加している．また，`token` アトム名には  $h_i$  と一意に対応が取れる整数値  $\langle h_i \rangle$  を付加している．

### 4.4.2 $\beta$ ルール

$\beta$  ルールは RETE の  $\beta$  ネットワークの処理を行うルールである．アトム  $h_i$  と  $h_j$  の組み合わせのトークンを生成するルールの一部を以下に示す．

$$\begin{aligned} & token\_ \langle h_i \rangle (!Ref_i), h_i\_rete(\dots, L_i, \dots, !Ref_i), \\ & mem\_ \langle h_j \rangle (!Ref_j), h_j\_rete(\dots, L_j, \dots, !Ref_j), \\ & L_i = L_j \\ & :- \\ & token\_ \langle h_i \rangle (!Ref_i), h_i\_rete(\dots, L_i, \dots, !Ref_j), \\ & joined\_mem\_ \langle h_j \rangle (!Ref_j), h_j\_rete(\dots, L_j, \dots, !Ref_j), \\ & L_i = L_j, \\ & token\_ \langle h_i h_j \rangle (!Ref_i, !Ref_j) \end{aligned}$$

このルールは  $h_i$  のトークンが Join ノードに到達したときの処理を表現している． $h_i$  のトークンが見つかったとき，Join ノードの反対側のメモリの要素と組み合わせを生成する．このとき， $h_i$  と  $h_j$  の間にリンクの接続関係がある場合，変数束縛の一致のテストと同様に  $L_i = L_j$  によって接続関係をテストしている．

また,  $h_i$  と  $h_j$  の間にあるガード条件もここでテストする.

$$\begin{aligned} & token\_ \langle h_i \rangle (!Ref_i), h_i\_rete(\dots, L_i, \dots, !Ref_i), \\ & mem\_ \langle h_j \rangle (!Ref_j), h_j\_rete(\dots, L_j, \dots, !Ref_j), \\ & \quad :- L_i ::= L_j \mid \\ & token\_ \langle h_i \rangle (!Ref_i), h_i\_rete(\dots, L_i, \dots, !Ref_j), \\ & joined\_mem\_ \langle h_j \rangle (!Ref_j), h_j\_rete(\dots, L_j, \dots, !Ref_j), \\ & token\_ \langle h_i h_j \rangle (!Ref_i, !Ref_j) \end{aligned}$$

上の例では  $L_i ::= L_j$  によって整数値が一致することをテストして, 条件を満たしたときのみ組み合わせのトークンを生成する.

#### 4.4.3 書換えルール

書換えルールは  $\alpha \cdot \beta$  ルールによって生成されたパターンマッチデータを選択して, 変換元ルールの右辺を実行するルールである.

$$\begin{aligned} & mem\_ \langle h_1 \dots h_n \rangle (!Ref_1, \dots, !Ref_n), h_1\_rete(\dots, !Ref_1), h_n\_rete(\dots, !Ref_n) \\ & \quad :- \\ & del(!Ref_1), \dots, del(!Ref_k), \\ & h_{k+1}\_rete(\dots, !Ref_{k+1}), h_n\_rete(\dots, !Ref_n), \\ & b_1, \dots, b_m \end{aligned}$$

$\alpha \cdot \beta$  ルールによってマッチするデータがあるときは  $mem\_ \langle h_1 \dots h_n \rangle (!Ref_1, \dots, !Ref_n)$  が存在する.  $del(!Ref_i)$  は削除するアトムを指定していて, このルールによってマッチしたアトム  $h_i$  を削除することを示している.  $h_{k+1}\_rete(\dots, !Ref_{k+1}), h_n\_rete(\dots, !Ref_n)$  はルールの左辺と右辺で変わらないアトムを表している.  $b_1, \dots, b_m$  は変換元ルールの右辺を生成することを示している.

#### 4.4.4 削除ルール

削除ルールはルールの実行によって削除されたアトムを RETE のメモリから削除するルールである.

$$\begin{aligned}
&del(!Ref), mem\_h_i(!Ref) :- del(!Ref). \\
&del(!Ref), mem\_h_i h_j(!Ref, -) :- del(!Ref). \\
&del(!Ref), mem\_h_i h_j(-, !Ref) :- del(!Ref). \\
&del(!Ref), mem\_h_i h_j h_k(!Ref, -, -) :- del(!Ref). \\
&del(!Ref), mem\_h_i h_j h_k(-, !Ref, -) :- del(!Ref). \\
&del(!Ref), mem\_h_i h_j h_k(-, -, !Ref) :- del(!Ref). \\
&\dots
\end{aligned}$$

第3章で述べたように負のトークンは効率が悪いので、ポインタ管理による削除を行う。削除すべきメモリは  $del(!Ref_i)$  の HyperLink によって接続されているため、HyperLink に繋がった全てのメモリアトムを削除することで実現できる。

## 4.5 RETE ルールの制御フロー

LMNtal プログラムを変換して得られる RETE ルールの制御フローを図 4.1 に示す。

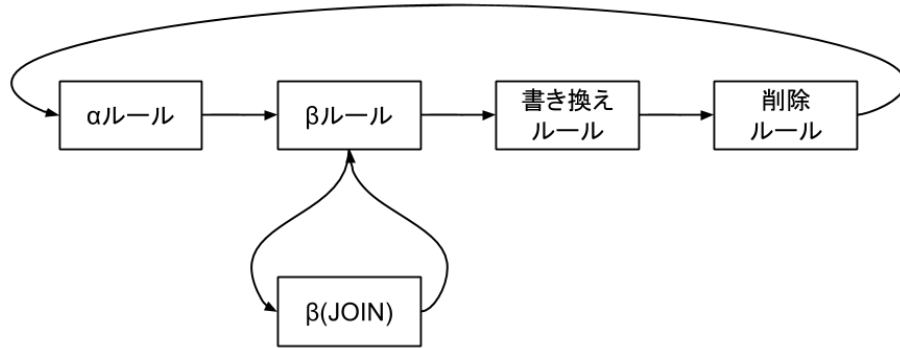


図 4.1 RETE ルールの制御フロー

処理系 SLIM は LMNtal のルールをプログラムファイルの上から順に適用しようとする。しかし、上の RETE ルールのようにルール適用の順序を逐次的にしたいことがある。こうした制御を可能とする LMNtal のプログラムテクニックを紹介する。

### 4.5.1 ルールの逐次実行

複数のルールを逐次実行する例を図 4.2 に示す. このプログラムは rule3,rule2,rule1 の順番に適用されて x(3) で終了する.

```

1 token1, x(1).
2 rule1 @@ token3, x($n) :- $m = $n - 1 | x($m).
3 rule2 @@ token2, x($n) :- $m = $n * 2 | token3, x($m).
4 rule3 @@ token1, x($n) :- $m = $n + 1 | token2, x($m).

```

図 4.2 ルールの逐次実行の例

### 4.5.2 ルール適用が停止したことの判定

LMNtal は拡張機能としてある膜内の全てのルールがもう適用できない状態を検出することができる. 膜内の全てのルールが適用し終わったら中のプロセスを膜の外に出す例を図 4.3 に示す.

```

1 {$p, @r}/ :- $p.

```

図 4.3 ルール停止条件の例

ルール停止条件とルールの逐次実行と組み合わせることで図 4.1 に示したフロー制御を実現することができる. フロー制御を行う LMNtal ルールを図 4.4 に示す. 1 行目のルールは  $\alpha$  ルールが全て適用されたら  $\beta$  ルールに遷移することを示す. 2-3 行目のルールはある 1 つのトークンに対して  $\beta$  ルールの Join を繰り返し適用して, トークンがなくなるまでループする. 全てのトークンについての Join が終了すると書き換えと削除ルールに遷移する. 5 行目のルールは書き換えと削除が全て適用されたら  $\alpha$  ルールへ遷移する. 6 行目のルールは書き換えるものがないときループを終了する.



1	$\{\$p[], @r, \text{alpha}\}/$	$:- \{\$p[], @r, \text{beta}\}.$
2	$\{\$p[], @r, \text{beta\_Join}\}/$	$:- \{\$p[], @r, \text{beta\_end}\}.$
3	$\{\$p[], @r, \text{beta\_end}\}/$	$:- \{\$p[], @r, \text{beta}\}.$
4	$\{\$p[], @r, \text{beta}\}/$	$:- \{\$p[], @r, \text{rewrite}\}.$
5	$\{\$p[], @r, \text{rewrite\_end}\}/$	$:- \{\$p[], @r, \text{alpha}\}.$
6	$\{\$p[], @r, \text{rewrite}\}/$	$:- \$p[].$

図 4.4 RETE ルールを制御するルール

## 4.6 RETE による uniq 制約の表現

LMNtal では拡張構文として `uniq` ガード条件 [6] が存在する. `uniq` ガード条件は LMNtal で CHR をエンコードするために導入された機能で, CHR の Propagation ルールと同じような機能を提供する. `uniq` の構文を以下に示す.

$$\text{Head} :- \text{uniq}(L_1, \dots, L_n) | \text{Body}.$$

このルールは  $L_1, \dots, L_n$  に束縛された同型なグラフ構造に対して高々 1 回のみ反応する.

この `uniq` の機能は 3.4 節で述べたプロダクションシステムのルール適用制御と同じであり, RETE によって表現可能である. プログラミング変換では, 書き換えルールにおいて `uniq` の対象であるアトムをメモリからのみ削除することで実現している.

## 第 5 章

# 評価実験

この章では，第 4 章で提案したプログラム変換手法に基づいて，いくつかの LMNtal プログラム変換したときの実行効率の評価について述べる．なおプログラムの変換は提案手法に基づき筆者が手作業で変換して評価した．実験環境を図 5.1 に示す．

表 5.1 実験環境

CPU	Intel Core i5-4690S @ 4x 3.9GHz
MEM	8GB
LMNtal コンパイラ	version 1.44
LMNtal オプション	<code>--slimcode -O3 --hl-opt</code>
SLIM	version 2.3.1
SLIM オプション	<code>--hl</code>

### 5.1 RAM マシンシミュレータに関する計測

RAM マシンシミュレータ [13] とは，図 5.1 に示すようなメモリ上の値を操作する命令列を解釈実行するマシンのシミュレーションプログラムのことである．図 5.1 は次に示すような数式を求める RAM マシン上の命令列である．

$$F(N) = F(N-2) * F(N-1)$$

$$F(0) = 1$$

$$F(1) = 1$$

i は命令を表し，第1引数がプログラム番地，第2引数が命令の種類，第3引数以降は命令のオペランドを表す．m はメモリを表し，第1引数がメモリ番地，第2引数が値を表す．c はプログラムカウンタを表す．

```

1  i(1,init,3).      % [3] 番目までの新しいメモリをで初期化する 0
2  i(2,imv,1,6).    % [6] = [[1]]
3  i(3,imv,2,7).    % [7] = [[2]]
4  i(4,mul,6,7).    % [7] = [7] * [6]
5  i(5,mvi,7,3).    % [[3]] = [7]
6  i(6,add,5,1).    % [1] = [1] + [5]
7  i(7,add,5,2).    % [2] = [2] + [5]
8  i(8,add,5,3).    % [3] = [3] + [5]
9  i(9,sub,5,4).    % [4] = [4] - [5]
10 i(10,cjmp,4,12). % PC = 12 (if [4] = 0)
11 i(11,jmp,1).     % PC = 1
12 i(12,halt).
13 m(1,8).
14 m(2,9).
15 m(3,10).
16 m(4,N).
17 m(5,1).
18 m(6,0).
19 m(7,0).
20 m(8,1).
21 m(9,1).
22 maxm(9).
23 c(1).

```

図 5.1 RAM マシンシミュレータの上で実行される命令列

この命令列とメモリとプログラムカウンタを受け取り実行をシミュレーションする LMNtal プログラムを図 5.2 に示す．

このプログラムのほとんどのルールは左辺と右辺で変化するアトムが c とひとつの m であり，ルール実行によるデータの変化が小さい．そのため，RETE によるパターンマッチのキャッシュが有効に働く例であると考えられる．

4 章で提案した変換手法によって図 5.2 のプログラムを変換したものを付録 A.1 に示す．また，比較のために同じプログラムを CHR で記述したものを付録 A.1 に示す．CHR のプログラムは SWI-Prolog (version 7.6.4) の上で実行を行った．これらの3つのプログラムを実行したときの実行時間による比較結果を図 5.3 に示す．凡例はそれぞれ LMNtal が変換元，RETE が変換後，CHR が SWI-prolog で同じ問題を記述したものを表す．

実験結果から変換元プログラムの実行時間オーダーが  $O(N^3)$  であったのに対し，変換後は  $O(N)$  となり実行効率が改善され N が 60 以降では実行時間が逆転していることがわかる．また，N が 10 から 50 の間に変換後の実行時間の方が大きいことがわかるが，これはプログラム変換によって変換元プログラムのルール数が 16 個であったのに対し，変

換後のルール数は 554 個に増大したためである。ルール数が増えることで1回のルールの適用の際に最悪の場合ルールの数だけ適用が試みられ、これがオーバーヘッドとなっていると考えられる。

```

1  add @@
2  i(L0,add,B0,A0), m(B1,Y), m(A1,X), c(L1) :-
3      L0:=L1, A0:=A1, B0:=B1, Z=X+Y, L_=L0+1 |
4      i(L0,add,B0,A0), m(B1,Y), m(A0,Z), c(L_).
5  sub @@
6  i(L0,sub,B0,A0), m(B1,Y), m(A1,X), c(L1) :-
7      L0:=L1, A0:=A1, B0:=B1, Z=X-Y, L_=L0+1 |
8      i(L0,sub,B0,A0), m(B1,Y), m(A0,Z), c(L_).
9  imv @@
10 i(L0,imv,B0,A0), m(B1,C0), m(C1,Y), m(A1,X), c(L1) :-
11     L0:=L1, A0:=A1, B0:=B1, C0:=C1, int(X), int(Y), L_=L0+1 |
12     i(L0,imv,B0,A0), m(B1,C0), m(C1,Y), m(A0,Y), c(L_).
13 jmp @@
14 i(L0,jmp,A), c(L1) :- L0:=L1, int(A) | i(L0,jmp,A), c(A).
15 cjmp @@
16 i(L0,cjmp,A0,J), m(A1,0), c(L1) :-
17     L0:=L1, A0:=A1, int(J) |
18     i(L0,cjmp,A0,J), m(A1,0), c(J).
19 cjmp @@
20 i(L0,cjmp,A0,J), m(A1,X), c(L1) :-
21     L0:=L1, A0:=A1, X=\=0, int(J), L_=L0+1 |
22     i(L0,cjmp,A0,J), m(A1,X), c(L_).
23 halt @@
24 i(L0,halt), c(L1) :- L0:=L1 | i(L0,halt).

```

図 5.2 RAM マシンシミュレータの LMNtal プログラム

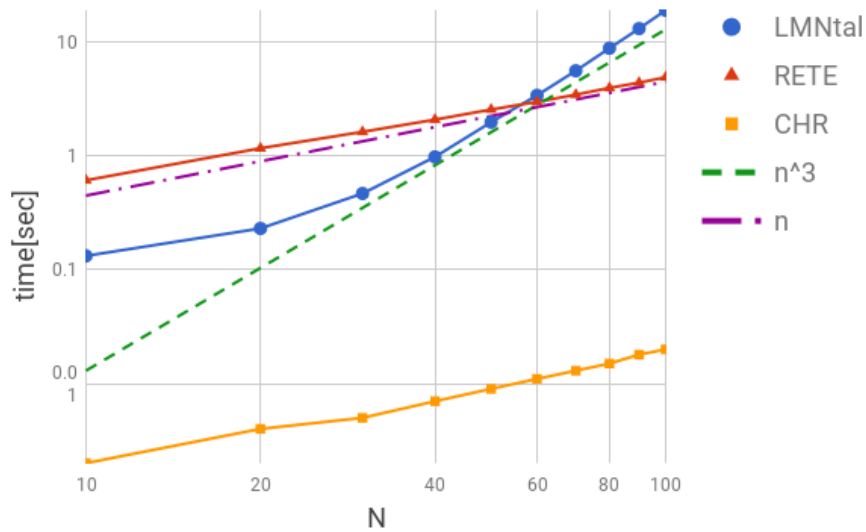


図 5.3 RAM マシンシミュレータの評価結果

## 5.2 最短経路問題に関する計測

最短経路問題はコスト付き経路のが与えられたとき、任意の頂点から全ての頂点へ到達するための経路の合計コストを算出する問題である。最短経路問題の LMNtal プログラムを図 5.5 に示す。このプログラムでは経路を 3 引数の path で表し、第 1 引数から第 2 引数への経路のコストが第 3 引数であることを表す。

このプログラムへの入力を図 5.2 に示す。この入力では右方向への一方通行な経路が生成され、各経路のコストはランダムに与えられる。m=3 で固定して n をスケールリングさせたときの実行時間の比較を図 5.6 に示す。

実行結果から変換後のプログラムの方が実行時間が短くなることがわかるが、オーダーは変化していないことがわかる。そこで、実行時間の他に実行される中間命令数について比較をしてみた。N が 4 (path の初期数が 36) のときの変換元プログラムと変換後のプログラムのルール実行の 1 ステップにかかる中間命令数を比較した結果を図 5.7 に示す。

図 5.7 の実行結果から、変換元のプログラムは後半になるに連れて中間命令数が増大していることがわかる。これは、プログラムの実行が進むにつれてグラフ全体における path アトムが増加することでルールにおける path の比較回数が増加していくためである。一方、変換後のプログラムの 1 ステップあたりに実行される中間命令数が常に

10000 個以下であることがわかる．これは，RETE によって path の組み合わせがキャッシュされ全体の path アトムの数によらずマッチングに成功するまでに必要な中間命令ステップ数がほぼ一定になったためであると考えられる．

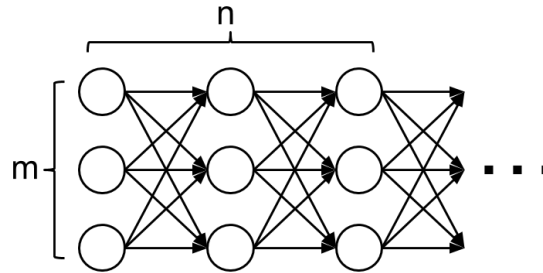


図 5.4 最短経路問題の入力

```

1 rem_long @@
2 path($x0,$y0,$a), path($x1,$y1,$b)
3   :- $x0==$x1, $y0==$y1, $a=<$b |
4 path($x0,$y0,$a).
5
6 path_add @@
7 path($x,$y0,$a), path($y1,$z,$b)
8   :- $y0==$y1, $x\==$z, $c=$a+$b, uniq($x,$y0,$y1,$z,$a,$b) |
9 path($x,$y0,$a), path($y1,$z,$b), path($x,$z,$c).

```

図 5.5 最短経路問題の LMNtal プログラム

### 5.3 バブルソートに関する計測

バブルソートのプログラムを図 5.8 に示す．また，長さ  $N$  のランダムなリストを与えたときの実行時間による比較結果を図 5.9 に示す．凡例 LMNtal(headatom) は LMNtal の headatom 中間命令 [16] を使って記述したバブルソートの実行時間結果である．headatom 命令は指定したアトムをそのアトムリストの先頭につなぎ換える命令である．ルールによって書き換わったアトムを headatom でアトムリストの先頭に挿入し，書き換わったアトムを優先的に findatom されるようにすることでパターンマッチの高速化

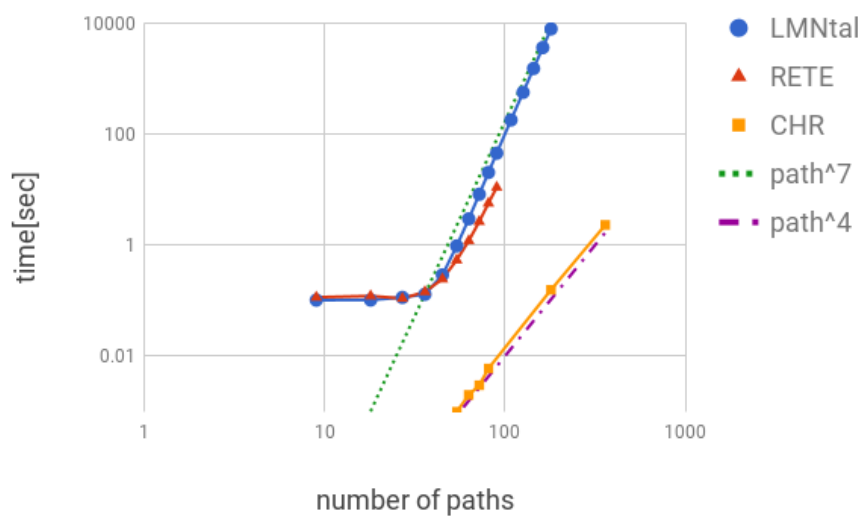


図 5.6 最短経路問題の評価結果

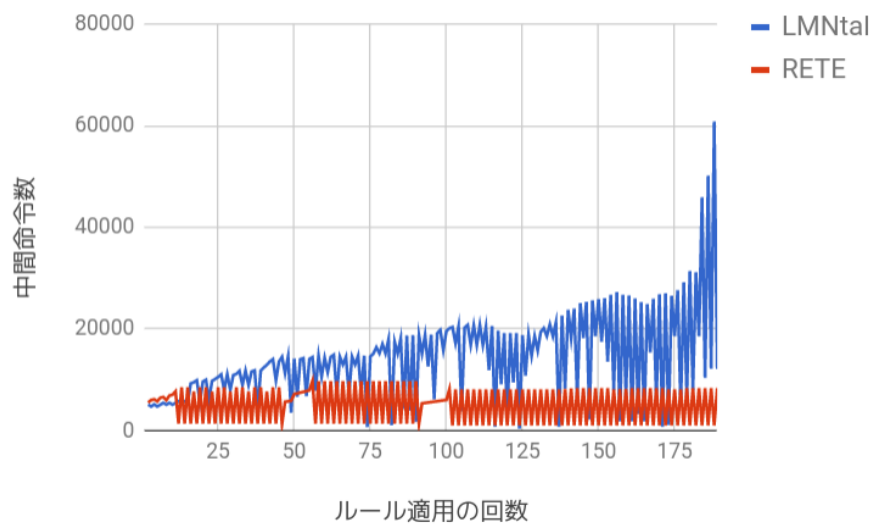


図 5.7 最短経路問題の中間命令数比較

を図る手法である。

図 5.9 の実行結果から、変換後の実行時間のオーダーは headatom 使用時と同じ  $O(N^2)$  であることがわかる。これはプログラム変換によって、データの差分を基点にマッチングを行うようになったため headatom と同じ効果が得られるようになったためであると考えられる。

```

1  sort @@
2  cons($x,L,H), cons($y,T,L) :- $x>$y | cons($y,L,H), cons($x,T,L).

```

図 5.8 バブルソートの LMNtal プログラム

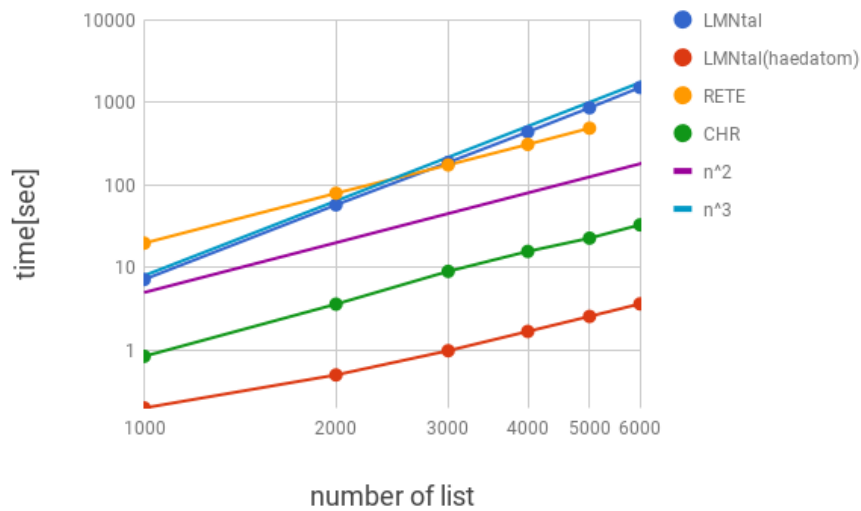


図 5.9 バブルソートの評価結果



## 第 6 章

# まとめと今後の課題

### 6.1 まとめ

本研究では LMNtal プログラムを RETE に基づくマッチングを行う LMNtal プログラムへ変換することによりグラフパターンマッチングの効率の改善を行った．実験により，ルール実行による変化が全体グラフに比べて小さいプログラムに対して実行時間オーダーが  $O(N^3)$  から  $O(N)$  に改善されることが確認できた．

### 6.2 今後の課題

#### 6.2.1 プログラムの自動変換

本研究では第 4 章で提案したプログラム変換手法に基づいてプログラムを手作業で変換したが，この自動化は必要である．プログラム変換の自動化に関する課題としては，変換によって得られるプログラムが複数通り考えられることが挙げられる．プログラムのルール集合に対応する RETE ネットワークは，Join ノードの組み合わせ方やネット枠ノードの共有の方法によって何通りものネットワーク構造が考えられる．さらに RETE ネットワーク構造によって実行効率が変化するため，最適な RETE ネットワーク構造を決定することは重要課題である．

プロダクションシステムの文献 [1] では，一度プロダクションシステムを実行して実行効率のプロファイリングをとり，プロファイリングを基に RETE ネットワーク構造に最適化を施すといった手法も提案されている．

### 6.2.2 プログラム変換以外の手法

本研究では LMNtal へ RETE を導入するために処理系 SLIM を拡張せずにプログラム変換手法を用いた。しかし、プログラム変換によってルール数が増大してしまい実行効率のオーバーヘッドの要因の 1 つであると考えられる。ルールが増加することによって効率が悪化する原因は、SLIM が定義されたルールを上から順に選択してパターンマッチを試みるためである。定義されたルールの上の方にパターンマッチに失敗しやすいルールが偏って存在すると、ルールの適用の度にパターンマッチの失敗が頻発して効率が悪くなる。オーバーヘッドを改善する手法として次の手法が考えられる。

#### 中間命令レベルで RETE を表現する

提案したプログラム変換手法では、 $\alpha$  ルール、 $\beta$  ルール、書換えルール、削除ルールを逐次的に実行させて RETE ネットワークのメモリの更新を表現している。この逐次制御は、 $\alpha$  ルールに含まれるルールが全て適用し終わったときに  $\beta$  ルールの適用に移るといような同期処理をルール上で表現している。しかし、例えば削除ルールが実行され続けている間でも  $\alpha$  ルールの定義が削除ルールより上にあるとき、 $\alpha$  ルールへのパターンマッチは試されて失敗している。つまり、逐次的なルールの制御は LMNtal プログラムで RETE を表現するために必要なものであって、LMNtal プログラムの効率面から見ると逐次制御は望ましくない。

そこで、SLIM の中間命令列レベルで RETE の逐次処理を表現する方法が考えられる。逐次制御されるルール集合の適用の始まりから適用できなくなるまでの実行ステップを中間命令列でひとまとめにできれば、他のルールへのパターンマッチ処理が挟まれない。これを実現するために利用できる loop 中間命令というものがある。loop 中間命令の仕様は次の通りである。

*loop*[[*instructions*...]]

引数に中間命令列 *instructions*... を受け取り繰り返し実行する命令。引数の中間命令列の実行中に失敗が発生したとき、loop の実行が終了して次の命令へ進む。

この loop 中間命令を使って  $\beta$  ルールの Join 演算と同じような処理を記述したものを図 6.1 に示す。この中間命令はアトム a が生成されたとき、アトム b との組み合わせを生成することを想定している。アトム a のトークンは 0 引数の *token\_a* アトムで表し、ア

トム b のメモリを 0 引数の `mem_b` アトムで表す。loop の 1 行目はアトム a の組み合わせとなるアトム b のメモリを探索する。探索に成功したら `mem_b` を探索済みを意味する `mem_b_joined` に書き換え、新しい組み合わせ `token_a_b` を生成する。そして `proceed` 命令まで実行したら再び loop 命令の 1 行目が始まる。これによって全ての `mem_b` との組み合わせ `token_a_b` を生成する処理を他のルールへのパターンマッチやルールの適用が挟まれずに実行できる。

このように loop 命令を駆使することで全ての Join 演算を中間命令上で表現することは可能であり、この処理は他のルールに邪魔されない。この手法を使った RETE の表現によるプログラムの実行効率の評価は今後の課題である。

```
1 newatom      [1, 0, 'token_a'_0]
2 loop [[
3     findatom  [2, 0, 'mem_b'_0]
4     removeatom [2, 0, 'mem_b'_0]
5     newatom    [3, 0, 'mem_b_joined'_0]
6     newatom    [4, 0, 'token_a_b'_0]
7     proceed   []
8 ]]
```

図 6.1 HyperLMNtal のプログラム例

## 謝辞

本研究を進めるにあたり様々な方の指導，助言をいただきました。まず，研究配属から今まで長きに渡ってご指導頂きました上田和紀教授に深く感謝致します。

また，3年間研究や研究以外でも一緒に助け合ってくれた同期の皆さんには感謝しております。特に研究の相談に常に親身になってくれて Archlinux の環境構築や Coq 勉強会などで巻き込み巻き込まれしてくれた恒川君や，ほぼ毎日お昼ご飯を一緒に食べつつ相談したり愚痴を聞いてくれたりしてくれた小山君には大変感謝しております。

最後に，入学から現在に至るまで，叱咤激励を私にかけてくれた家族に深く感謝致します。

2018 年 1 月 松澤 望

## 参考文献

- [1] 石田亨. プロダクションシステムの発展. 朝倉書店, 1996.
- [2] Robert B Doorenbos. Production matching for large learning systems. Technical Report CMU-CS-95-113, Carnegie-Mellon University, 1995.
- [3] 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, pp. 126–142, 2004.
- [4] 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, pp. 247–277, 2008.
- [5] Kazunori Ueda and Seiji Ogawa. HyperLMNtal: An extension of a hierarchical graph rewriting model. *KI - Künstliche Intelligenz*, Vol. 26, No. 1, pp. 27–36, Feb 2012.
- [6] 小川誠司, 目黒学, 上田和紀. 階層グラフ書換えモデルを拡張した HyperLMNtal の実現. 人工知能学会全国大会論文集, Vol. 25, No. 2I1-4, 2011.
- [7] Charles L Forgy. Ops5 user’s manual. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.
- [8] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, Vol. 19, No. 1, pp. 17–37, 1982.
- [9] Clips. <http://www.clipsrules.net/>.
- [10] Jess® the rule engine for the java™ platform. <http://www.jessrules.com/jess/>.
- [11] Daniel P Miranker. *TREAT: A new and efficient match algorithm for AI production system*. Morgan Kaufmann, 1990.
- [12] Daniel P Miranker, David A Brant, Bernie J Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *Proc. AAAI’90*, pp. 685–692, 1990.
- [13] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and

- Complexity of Constraint Handling Rules. *ACM Trans. Program. Lang. Syst.*, Vol. 31, No. 2, pp. 8:1–8:42, 2009.
- [14] Peter Van Weert, Tom Schrijvers, and Bart Demoen. KU Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *Proceedings of the 2nd Workshop on Constraint Handling Rules*, pp. 47–62. Department of Computer Science, KU Leuven, 2005.
- [15] Peter Van Weert. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 22, No. 11, pp. 1521–1534, 2010.
- [16] 青山龍一. グラフ書換え系 LMNtal の実行時処理系 SLIM におけるグラフパターンマッチングの高速化. Master's thesis, 早稲田大学大学院 基幹理工学研究科 情報理工学専攻, 2015.

## 外部発表

- [1] 松澤望, 上田和紀 : グラフ照合のキャッシュ化による階層グラフ書換え言語 LMNtal の高速化, PPL2017 (ポスター発表).
- [2] 松澤望, 上田和紀 : Incremental Pattern Matching による階層グラフ書換え言語 LMNtal の高速化, 情報処理学会第 80 回全国大会, 2018 (発表予定).

# Appendix

第 5 章で評価したプログラムファイルを GitLab リポジトリ (<https://matocq.ueda.info.waseda.ac.jp/gitlab/matsuzawa/translate-lmntal-to-rete/tree/master/sample>) で公開している. \*.lmn が変換元のプログラム, \*.rete.lmn が変換後のプログラム, \*.pl が CHR で書かれたプログラムである.

## A.1 RAM マシンシミュレータ

### CHR プログラム

```

1  /*
2  RAM Simulator
3  Reference :
4  Jon Sneyers, Tom Schrijvers, and Bart Demoen. 2009. The computational power and
    complexity of constraint handling rules. ACM Trans. Program. Lang. Syst. 31, 2,
    Article 8 (February 2009), 42 pages.
5  */
6
7  :- module(ram_sim,[i/4, i/3, i/2, m/2, c/1, maxm/1, initm/3, input/1]).
8  :- use_module(library(chr)).
9  %% :- chr_constraint i(+int,+,+int,+int), i(+int,+,+int), i(+int,+), m(+int,+int), c(+int
    ), maxm(+int), initm(+int,+int,+int), input(+int).
10 :- chr_constraint i/4, i/3, i/2, m/2, c/1, maxm/1, initm/3, input/1.
11 :- chr_option(optimize,full).
12
13 %% init @ i(L,init,A), m(A,B) \ maxm(M), c(L) <=> M_ is M+1 | initm(M_,B,L).
14
15 %% initm1 @ initm(A,B,L) <=> A=<B, A_ is A+1 | m(A,O), initm(A_,B,L).
16 %% initm2 @ initm(A,B,L), m(B,_ ) <=> A>B, L_ is L+1 | m(B,O), maxm(B), c(L_).
17
18 add @ i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> Z is X+Y, L_ is L+1 | m(A,Z), c(L_).
19 sub @ i(L,sub,B,A), m(B,Y) \ m(A,X), c(L) <=> Z is X-Y, L_ is L+1 | m(A,Z), c(L_).
20 %% mul @ i(L,mul,B,A), m(B,Y) \ m(A,X), c(L) <=> Z is X*Y, L_ is L+1 | m(A,Z), c(L_).
21
22 %% imv @ i(L,imv,B,A), m(B,C), m(C,Y) \ m(A,_ ), c(L) <=> L_ is L+1 | m(A,Y), c(L_).
23 %% mvi @ i(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,_ ), c(L) <=> L_ is L+1 | m(C,Y), c(L_).
24
25 jmp @ i(L,jmp,A) \ c(L) <=> c(A).
26
27 cjmp @ i(L,cjmp,A,J), m(A,O) \ c(L) <=> c(J).
28 cjmp @ i(L,cjmp,A,_ ), m(A,X) \ c(L) <=> X\=0, L_ is L+1 | c(L_).
29
30 halt @ i(L,halt) \ c(L) <=> true .

```



```

31
32 input(N) <=>
33   i(1,init,3),      % [3] 番目までの新しいメモリをで初期化する 0
34   i(2,imv,1,6),    % [6] = [[1]]
35   i(3,imv,2,7),    % [7] = [[2]]
36   i(4,mul,6,7),     % [7] = [7] * [6]
37   i(5,mvi,7,3),     % [[3]] = [7]
38   i(6,add,5,1),     % [1] = [1] + [5]
39   i(7,add,5,2),     % [2] = [2] + [5]
40   i(8,add,5,3),     % [3] = [3] + [5]
41   i(9,sub,5,4),     % [4] = [4] - [5]
42   i(10,cjmp,4,12),  % PC = 12 (if [4] = 0)
43   i(11,jmp,1),      % PC = 1
44   i(12,halt),
45   m(1,8),
46   m(2,9),
47   m(3,10),
48   m(4,N), % m(4,N), N can be scale
49   m(5,1),
50   m(6,0),
51   m(7,0),
52   m(8,1), % F(0)
53   m(9,1), % F(1)
54   maxm(9),
55   c(1).

```

### 変換後 LMNtal プログラム

ファイルサイズが大きいため GitLab リポジトリの `ram_sim.rete.lmn` ファイルを参照すること。

## A.2 最短経路問題

### CHR プログラム

```

1  :- module(shortestPath,[path/3, input/1]).
2  :- use_module(library(chr)).
3  :- chr_constraint path(+,+,+int), input/1.
4  :- chr_option(optimize,full).
5
6  rem_long @ path(X,Y,A) \ path(X,Y,B) <=> A =< B      | true.
7  path_add @ path(X,Y,A) , path(Y,Z,B) ==> X \= Z, C is A+B | path(X,Z,C).

```

### 変換後 LMNtal プログラム

```

1  /*
2  * Shortest path problem
3  * translated from shortestPath-chr.lmn to RETE
4  *
5  * Usage
6  *   echo <num> | slim -I./module --hl shortestPath-chr.rete.lmn
7  */
8
9  transA2B    @@ {$p[], @r, alpha}/      :- {$p[], @r, beta}.

```

```

10 transBj2Be @@ {$p[], @r, beta_Join}/      :- {$p[], @r, beta_end}.
11 transBe2B  @@ {$p[], @r, beta_end}/      :- {$p[], @r, beta}.
12 transB2R   @@ {$p[], @r, beta}/          :- {$p[], @r, rewrite}.
13 transRe2A  @@ {$p[], @r, rewrite_end}/    :- {$p[], @r, alpha}.
14
15 transAbort @@ {$p[], @r, rewrite}/        :- $p[].
16 mem_0(!R) :- .
17 mem_1(!R) :- .
18 path_rete(SRC,DST,COST,!R) :- path(SRC,DST,COST).
19
20 {
21   init({
22     util.use.
23
24     input @@
25     RET = input_from_stdin
26     :- RET = util.of_str(util.read_line(util.stdin, P)), util.free_port(util.
      close_port(P)), util.srnd(0).
27
28     gen(3,input_from_stdin,100).
29
30     gen @@
31     gen($h,$w,C) :- $l=$h*($w-1) | input(1,$h,$l,C).
32
33     gen @@
34     input($n,$m,$l,$c)
35     :- $n < $l, $e0=$n+$m-1, $s1=$n+$m, $e1=$n+$m*2-1, $n_=$n+$m, int($c) |
36     path(util.set($n,$e0), util.set($s1,$e1), util.rnd($c)), input($n_, $m, $l, $c).
37
38     finish @@
39     input($n,$m,$l,$c) :- $n>$l, int($m), int($c) | .
40     alpha.
41   }).
42   init @@ init({$p[], @r}/) :- $p[].
43
44   % =====
45   % alpha net (rewrite_end -> alpha)
46   % =====
47   find_path @@
48   alpha, path(SRC,DST,COST)
49   :-
50   alpha, path_rete(SRC,DST,COST,!Ref), token_0(!Ref), token_1(!Ref).
51
52   % =====
53   % beta net (alpha -> beta) (beta_end -> beta)
54   % =====
55   activate0 @@ beta, token_0(Ref) :- beta_Join, active_token_0(Ref).
56   activate1 @@ beta, token_1(Ref) :- beta_Join, active_token_1(Ref).
57
58   join0_1_l @@
59   beta_Join,
60   active_token_0(!RefT), path_rete(L0i,L1i,L2i,!RefT), mem_1(!RefM), path_rete(L0j,L1j,
     L2j,!RefM)
61   :- L0i==L0j, L1i==L1j, L2i=<L2j |
62   beta_Join,
63   active_token_0(!RefT), path_rete(L0i,L1i,L2i,!RefT), joined_mem_1(!RefM), path_rete(
     L0j,L1j,L2j,!RefM),
64   mem_0_1_rm(!RefT,!RefM).
65
66   join0_1_r @@
67   beta_Join,
68   active_token_1(!RefT), path_rete(L0j,L1j,L2j,!RefT), mem_0(!RefM), path_rete(L0i,L1i,
     L2i,!RefM)
69   :- L0i==L0j, L1i==L1j, L2i=<L2j |
70   beta_Join,

```

```

71     active_token_1(!RefT), path_rete(L0j,L1j,L2j,!RefT), joined_mem_0(!RefM), path_rete(
72         L0i,L1i,L2i,!RefM),
73     mem_0_1_rm(!RefM,!RefT).
74
75     join0_1_l @@
76     beta_Join,
77     active_token_0(!RefT), path_rete(L0i,L1i,L2i,!RefT), mem_1(!RefM), path_rete(L0j,L1j,
78         L2j,!RefM)
79     :- L1i==L0j, L0i\==L1j |
80     beta_Join,
81     active_token_0(!RefT), path_rete(L0i,L1i,L2i,!RefT), joined_mem_1(!RefM), path_rete(
82         L0j,L1j,L2j,!RefM),
83     mem_0_1_ad(!RefT,!RefM).
84
85     join0_1_r @@
86     beta_Join,
87     active_token_1(!RefT), path_rete(L0j,L1j,L2j,!RefT), mem_0(!RefM), path_rete(L0i,L1i,
88         L2i,!RefM)
89     :- L1i==L0j, L0i\==L1j |
90     beta_Join,
91     active_token_1(!RefT), path_rete(L0j,L1j,L2j,!RefT), joined_mem_0(!RefM), path_rete(
92         L0i,L1i,L2i,!RefM),
93     mem_0_1_ad(!RefM,!RefT).
94
95     % =====
96     % beta net ending (beta_Join -> beta_end)
97     % =====
98     deactivate0 @@ beta_end, active_token_0(Ref) :- beta_end, mem_0(Ref).
99     deactivate1 @@ beta_end, active_token_1(Ref) :- beta_end, mem_1(Ref).
100
101     rmJoined0 @@ beta_end, joined_mem_0(Ref) :- beta_end, mem_0(Ref).
102     rmJoined0 @@ beta_end, joined_mem_1(Ref) :- beta_end, mem_1(Ref).
103
104     % =====
105     % rewrite (beta -> rewrite)
106     % =====
107     % rem_long @@
108     rem_log @@
109     rewrite, mem_0_1_rm(!Ref0,!Ref1), path_rete($x1,$y1,$b,!Ref1)
110     :- unary($x1), unary($y1), int($b) |
111     rewrite_end, del_wme(!Ref1).
112
113     % path_add @@
114     path_add @@
115     rewrite, mem_0_1_ad(!Ref0,!Ref1), path_rete($x,$y0,$a,!Ref0), path_rete($y1,$z,$b,!
116         Ref1)
117     :- unary($x), unary($y0), int($a), unary($y1), unary($z), int($b), $c=$a+$b |
118     rewrite_end, path_rete($x,$y0,$a,!Ref0), path_rete($y1,$z,$b,!Ref1), path($x,$z,$c).
119
120     del_wme0 @@ rewrite_end, del_wme(!Ref), mem_0(!Ref) :- rewrite_end, del_wme(!Ref).
121     del_wme1 @@ rewrite_end, del_wme(!Ref), mem_1(!Ref) :- rewrite_end, del_wme(!Ref).
122     del_wme0_1 @@ rewrite_end, del_wme(!Ref0), mem_0_1_rm(!Ref0,!Ref1) :- rewrite_end,
123         del_wme(!Ref0).
124     del_wme0_1 @@ rewrite_end, del_wme(!Ref0), mem_0_1_rm(!Ref1,!Ref0) :- rewrite_end,
125         del_wme(!Ref0).
126     del_wme0_1 @@ rewrite_end, del_wme(!Ref0), mem_0_1_ad(!Ref0,!Ref1) :- rewrite_end,
127         del_wme(!Ref0).
128     del_wme0_1 @@ rewrite_end, del_wme(!Ref0), mem_0_1_ad(!Ref1,!Ref0) :- rewrite_end,
129         del_wme(!Ref0).
130
131     del_wme_end @@ rewrite_end, del_wme(!Ref) :- num(!Ref,N), N==1 | rewrite_end.
132 }.
```

## A.3 バブルソート

### CHR プログラム

```

1  :- module(bubble,[cons/3, n/2, input/1]).
2  :- use_module(library(chr)).
3  :- chr_constraint cons(+int,?,?), n(+int,?), input(+int).
4  :- chr_option(optimize,full).
5
6  sort @ cons(X,L,H), cons(Y,T,L) <=> X>Y | cons(Y,L,H), cons(X,T,L).
7
8  gen @ n(N,L) <=> N>0, N_ is N-1, random(0,1000,M) | n(N_,P), cons(M,L,P).
9  input(N) <=> n(N,nil).
```

### 変換後 LMNtal プログラム

```

1  /*
2  * Bubble sort
3  * translated from bubble.lmn to RETE
4  *
5  * Usage
6  *   echo <num> | slim -I./module --hl bubble.rete.lmn
7  */
8
9  transA2B   @@ {$p[], @r, alpha}/      :- {$p[], @r, beta}.
10 transBj2Be @@ {$p[], @r, beta_Join}/   :- {$p[], @r, beta_end}.
11 transBe2B  @@ {$p[], @r, beta_end}/    :- {$p[], @r, beta}.
12 transB2R   @@ {$p[], @r, beta}/        :- {$p[], @r, rewrite}.
13 transRe2A  @@ {$p[], @r, rewrite_end}/  :- {$p[], @r, alpha}.
14
15 transAbort @@ {$p[], @r, rewrite}/      :- $p[].
16 mem_0(!R) :- .
17 mem_1(!R) :- .
18 cons(X,H,L,!R) :- cons(X,H,L).
19
20 {
21   init({
22     util.use.
23
24     input @@
25     RET = input_from_stdin :- RET = util.of_str(util.read_line(util.stdin, P)), util.
26     free_port(util.close_port(P)), util.srnd(0).
27
28     % ランダムリストの生成
29     gen0 @@ n(N,L) :- N>0, M=N-1 | n(M,cons(util.rnd(1000),L)). % ret=[rnd,rnd,rnd
30     ...,]
31
32     % 昇順に揃ったリストの生成
33     % gen0 @@ n(N,L) :- N>0, M=N-1 | n(M,cons(N,L)). % ret=[1,2,3,...]
34
35     % 降順に揃ったリストの生成
36     % gen0 @@ n(N,L) :- N>0, M=N-1 | cons(N,n(M),L). % ret=[N,N-1,N-2,...]
37
38     % ret=[1,2,3,...] の途中の数個について隣と入れ替えたリストの生成
39     % flag(100).
40     % gen0 @@ n(N,L), flag(R) :- N>0, M=N-1, R < 3 | n(M,rcons(N,L)), flag(util.rnd
41     (100)).
42     % gen1 @@ n(N,L), flag(R) :- N>0, M=N-1, R >= 3 | n(M, cons(N,L)), flag(util.rnd
43     (100)).
```

```

40      % gen2 @@ rcons(X,L,H), cons(Y,T,L) :- cons(Y,L,H), cons(X,T,L).
41
42      n(input_from_stdin,d).
43      alpha.
44  }).
45  init@@init({$p[], @r}/) :- $p[].
46
47  % =====
48  % alpha net (rewrite_end -> alpha)
49  % =====
50  find_dot @@
51  alpha, cons(L0,L1,L2) :- alpha, cons(L0,L1,L2,!Ref), token_0(!Ref), token_1(!Ref).
52
53  % =====
54  % beta net (alpha -> beta) (beta_end -> beta)
55  % =====
56  activate0 @@ beta, token_0(Ref) :- beta_Join, active_token_0(Ref).
57  activate1 @@ beta, token_1(Ref) :- beta_Join, active_token_1(Ref).
58
59  join0_1_l @@
60  beta_Join,
61  active_token_0(!RefT), cons(T_L0,T_L1,T_L2,!RefT), mem_1(!RefM), cons(M_L0,M_L1,M_L2
62    ,!RefM), T_L1=M_L2
63    :- T_L0>M_L0 |
64  beta_Join,
65  active_token_0(!RefT), cons(T_L0,T_L1,T_L2,!RefT), joined_mem_1(!RefM), cons(M_L0,
66    M_L1,M_L2,!RefM), T_L1=M_L2,
67  mem_0_1(!RefT, !RefM).
68
69  join0_1_r @@
70  beta_Join,
71  active_token_1(!RefT), cons(T_L0,T_L1,T_L2,!RefT), mem_0(!RefM), cons(M_L0,M_L1,M_L2
72    ,!RefM), T_L2=M_L1
73    :- M_L0>T_L0 |
74  beta_Join,
75  active_token_1(!RefT), cons(T_L0,T_L1,T_L2,!RefT), joined_mem_0(!RefM), cons(M_L0,
76    M_L1,M_L2,!RefM), T_L2=M_L1,
77  mem_0_1(!RefM, !RefT).
78
79  % =====
80  % beta net ending (beta_Join -> beta_end)
81  % =====
82  deactivate0 @@ beta_end, active_token_0(Ref) :- beta_end, mem_0(Ref).
83  deactivate1 @@ beta_end, active_token_1(Ref) :- beta_end, mem_1(Ref).
84
85  rmJoined0 @@ beta_end, joined_mem_0(Ref) :- beta_end, mem_0(Ref).
86  rmJoined1 @@ beta_end, joined_mem_1(Ref) :- beta_end, mem_1(Ref).
87
88  % =====
89  % rewrite (beta -> rewrite)
90  % =====
91  sort @@
92  rewrite, mem_0_1(!Ref0,!Ref1), cons(Ref0_L0,Ref0_L1,Ref0_L2,!Ref0), cons(Ref1_L0,
93    Ref1_L1,Ref1_L2,!Ref1)
94    :-
95  rewrite_end, cons(Ref1_L0,Ref0_L1,Ref0_L2), cons(Ref0_L0,Ref1_L1,Ref1_L2), del_wme(!
96    Ref0), del_wme(!Ref1).
97
98  del_wme0 @@
99  rewrite_end, del_wme(!Ref), mem_0(!Ref)
100    :-
101  rewrite_end, del_wme(!Ref).
102
103  del_wme1 @@
104  rewrite_end, del_wme(!Ref), mem_1(!Ref)

```

```
99      :-
100      rewrite_end, del_wme(!Ref).
101
102      del_wme0_1 @@
103      rewrite_end, del_wme(!Ref0), mem_0_1(!Ref0, !Ref1)
104      :-
105      rewrite_end, del_wme(!Ref0).
106
107      del_wme0_1 @@
108      rewrite_end, del_wme(!Ref0), mem_0_1(!Ref1, !Ref0)
109      :-
110      rewrite_end, del_wme(!Ref0).
111
112      del_wme_end @@ rewrite_end, del_wme(!Ref) :- num(!Ref,N), N:=1 | rewrite_end.
113  }.
```