# グラフ書換え言語LMNtalからのCプログラムの自動生成

冨岡 太一　上田 和紀

手続き型プログラムではポインタを用いることで，リストや木のような代数的データ構造よりも複雑な，グラフ構造を持つデータを扱うことができる．グラフ書換え言語ではこういった複雑なデータ構造の正当性を保つ操作を容易に記述できる．しかしながら，グラフ書換え言語で記述されたプログラムは一般に手続き型プログラムより性能が低く，またその成果を既存の言語と連携して使用することも難しい．この研究では，グラフ書換え言語 LMNtal で書かれたプログラムを C などのより低水準な手続き型言語で書かれた高速なプログラムへと変換することで，グラフ書換えに基づいたプログラムをより効率的でユーザが利用しやすい形で得ることを目的とする．本論文では LMNtal のサブセット言語を定義し，その言語を C へと変換するためのアルゴリズムを示す．また，いくつかの例題を用いてその有用性を検証する．

## 1　はじめに

オブジェクトとオブジェクト同士の参照からなるデータ構造への操作はあらゆるプログラミングパラダイムにおいて要求される．しかし，データ構造への操作によってデータ構造が不正にならないように手続き型言語でプログラムを書くことは容易ではない．データ構造が複雑になればなるほど操作の記述は複雑になる．それに加えてダングリングポインタや同じオブジェクトを指す複数の参照が存在するときなど，考慮しなければならない問題は多い．

こういった問題に対し，プログラムがデータ構造の操作を適切に行うことを保証あるいは検証するための手法が考えられてきた．関数型言語ではデータ構造を型として定義する．それによってプログラムの入力と出力となるデータの構造が適切であることを保証できる．その他にも，参照を扱うプログラムを検証する手法に分離論理 [13] がある．分離論理ではヒープと参照を形式化することによって参照を扱うプログラムを形式的に記述し検証することを可能にしている．

しかし，依然としてこれらの手法のみでは容易には扱えないプログラムが存在する．多くの関数型言語が備えている，代数的データ構造を扱う型システムでは根がちょうど 1 つある木構造は上手く扱うことができる．一方，根が複数本ある構造や循環などを持つ構造を扱うことは難しい．分離論理に基づいた検証でも，データ構造の仕様記述はヒープと参照を具体的に書き下さなければならず，ユーザにとっての負担は大きい．最近では分離論理によるプログラムの自動検証もされている [3] が，未だ研究途上である．

一般に，木よりも複雑な構造はグラフを形成する．グラフ構造を直接扱う計算モデルとしてはグラフ書換え系がある．グラフ書換え系には AGG [6] や PROGRES [15]，我々の実装である LMNtal [18] [17] など多くの処理系が存在する．グラフ書換え系は 1 つ以上の書換え規則からなる計算モデルであり，書換え規則をグラフに適用することによって計算を行う．書換え規則は 2 つのグラフパターンを左右に並べた形で表

---

* Automatic Generation of C Programs from Graph Rewriting Language LMNtal

Taichi Tomioka, 早稲田大学大学院基幹理工学研究科情報理工・情報通信専攻, Dept. of Computer Science and Enginerring, WasedaUniversity.
Kazunori Ueda, 早稲田大学理工学術院情報理工学科, Dept. of Information and Computer Science, Waseda University.

現される．書換え規則の適用とは，左辺のグラフパターンと一致する部分グラフを右辺のグラフパターンに置き換えることである．グラフ書換え系はグラフの構造を直接扱うことができる．しかし，グラフのパターンマッチは一般には同型サブグラフの探索を必要とするため計算コストがかかる．また，手続き型言語との隔たりも大きいため，グラフ書換え系での成果を現在ソフトウェア開発の主流である手続き型言語へと活かすことは実用上難しい．

本研究では，手続き型言語において複雑なデータ構造を扱うことの難しさとグラフ書換え系の扱いにくさを解決するために，グラフ書換え言語 LMNtal のプログラムから手続き型言語 C のプログラムを生成する手法を提案し実装する．本論文では LMNtal と C の橋渡しとなる言語 ADLMNtal の定義と，一部制約を持つ ADLMNtal プログラムの C への変換手法を提案，実装した．この実装は概念実証を目的としているため，細かな最適化は行わずに基本となる部分を優先的に実装した．実装は LMNtal で記述され，全部で約 800 本の書換え規則からなる．複数の例題に対して提案手法が正しく動くことを確認し，効率化，C 言語との連携の 2 つの目的を達成した．

## 2 関連研究

グラフ書換え系から手続き型プログラムを生成する研究やツールは著者の知る限りいくつか存在する．

文献 [2] はグラフ書換えモデルの 1 つである GP2 [10] [11] から同様の操作を行う C プログラムを生成する研究である．GP2 プログラム中に出現する書換え規則を C のコード片にそれぞれ対応づけることによって変換を行う．書換え規則を静的に解析することで最適化された C プログラムを出力し，高速な書換えを可能にしている．

グラフ書換え言語の 1 つである PROGRES（PROgramming with Graph REwriting Systems）[15] は強い型付きのマルチパラダイム言語である．PROGRES にはグラフィカルな開発環境が提供されている．開発環境で記述したプログラムはそのまま解釈して実行することも可能である一方，Modula-2 と C のプログラムにコンパイルすることができる [16]．コンパイルされたプログラムはランタイムライブラリが提供する API によってより大きなプログラムへと組み込むことができる．

また，モデル駆動型アーキテクチャ [8] にも同様の手法が存在する．モデル駆動型アーキテクチャで記述される可視化されたモデルはグラフ書換え系と相性が良い．そこで UML（Unified Modeling Language）[4] や MOF（MetaObject Facility）[9] などのモデリング言語で記述されたモデルについてグラフ書換え系を与えることでモデルと実装との差異を埋めることができる [20]．GReAT [1] [5] はそのような目的で開発された言語である．そのコンパイラである CG（Code Generator）は C++のプログラムを出力する [20]．出力されたプログラムはランタイムライブラリ上で動作する．

上述した研究ではいずれもグラフを表現するための抽象データ型やそれを扱うライブラリを必要とする．本研究では生成されたソースコードをユーザがある程度理解して修正することを意図しているため，データ構造に関しては参照を表す型以外は通常の C プログラミングと近い形で記述されている．また，グラフすべてをデータモデルとして扱うのではなく，グラフのうちいくつかの節点が操作を表すものと考えることでマッチング処理を簡略化した．

## 3 グラフ書換え言語 LMNtal

本節ではグラフ書換え言語 LMNtal とそのサブセット言語である FlatLMNtal について簡単に述べる．詳細については文献 [18] [17] を参照してほしい．

LMNtal プログラムはアトム，リンク，膜，ルールの基本要素からなる．アトム，リンクは直感的にはグラフ理論における節点と辺に対応している．LMNtal ではアトム，リンクを用いてプロセスと呼ばれる無向多重グラフを表現する．ただし，各アトムの辺は順序を持つため，正確には無向グラフではない．

LMNtal の構文の一部を図 1 に示す．$p(X_1, \ldots, X_m)$ と書くと，$p$ がアトム，$(X_1, \ldots, X_m)$ がそのアトムに接続しているリンクを表す．リンクは 2 つのアトムを接続するものであるから，1 つのグラフ表現中には同じリンク名は高々2 つしか書くこ

$$
\begin{array}{lcll}
P & ::= & 0 & \text{(空)} \\
 & | & p(X_1,\ldots,X_m) \quad (m \geq 0) & \text{(アトム)} \\
 & | & P,P & \text{(分子)} \\
 & | & \{P\} & \text{(セル)} \\
 & | & T\text{:-}T & \text{(ルール)} \\
T & ::= & 0 & \text{(空)} \\
 & | & p(X_1,\ldots,X_m) \quad (m \geq 0) & \text{(アトム)} \\
 & | & T,T & \text{(分子)} \\
 & | & \{T\} & \text{(セル)} \\
 & | & T\text{:-}T & \text{(ルール)}
\end{array}
$$

**図 1　LMNtal の基本構文**

```
R = append(nil, Y)        :- R = Y.
R = append(cons(A, B), Y) :-
   R = cons(A, append(B, Y)).
```

**図 2　LMNtal ルールの例**

とができない．特別なアトムとして，アトム'='は $X$=$Y$ と表記することができ，リンク $X$ と $Y$ とを接続する．また，記述を簡単にするために，$p(s_1,\ldots,s_m), q(t_1,\ldots,t_n)$ は $s_m$ と $t_i$ が同じリンクならば $q(t_1,\ldots,t_{i-1},p(s_1,\ldots,s_{m-1}),t_{i+1},\ldots,t_n)$ と略記してよいとする．{} は膜と呼ばれ，LMNtal においてプロセスをグループ化したセルを構成するために用いられる．膜は入れ子にすることができ，階層構造を形成する．膜を含まない LMNtal のサブセット言語は FlatLMNtal と呼ばれる．

ルールは左辺のグラフパターンを右辺のグラフパターンに書き換える書換え規則である．例えば，図 2 に示す 2 本のルールがあるとする．このルールのうち 2 本目のルールは図 3 の左側にマッチするグラフを右側の対応するグラフに書き換える．ルールの両辺に 1 度ずつ出現するリンクはどのアトムにつながっていてもよい．ルールに出現するリンクのうち，左辺と右辺のアトムの引数を繋ぐリンクを自由リンク，そうでないものを局所リンクという．

図 2 の 2 本のルールは LMNtal におけるリストを


**図 3　LMNtal の書換え規則**

連結する．LMNtal では無向グラフによってリストを表現するため，これは他の言語では双方向リンクリストに相当する．nil，cons はリストの構造を表すためのアトムである．2 つの書換え規則を適用することによって，append アトムの 1 番目と 2 番目のリンクにつながったリストは連結され 3 番目のリンクへと接続される．

## 4　LMNtal サブセット ADLMNtal

本節ではさらに，LMNtal を手続き型プログラムへと変換しやすくするためにサブセット言語 ADLMNtal を定義する．本論文では実装の都合上 LMNtal サブセットである FlatLMNtal のサブセットとして ADLMNtal を定義するが，形式的には膜を含むように定義することも可能である．さらに，任意の FlatLMNtal プログラムが ADLMNtal へと変換できることを示すことで ADLMNtal が FlatLMNtal と同等の表現力を持つことを示す．

### 4.1　ADLMNtal

ADLMNtal は FlatLMNtal のアトムを 2 種類に分類することで得られる．それらのアトムをアクティブアトム（Active atom）とデータアトム（Data atom）と呼ぶ．ADLMNtal の構文を図 4 に示す．また，これ以降有限個のリンクの列 $X_1,\ldots,X_m$ を飾り文字を用いて $\mathcal{X}$ と表す．

注目するべきは書換え規則の構文である．ここでは書換え規則の左辺には必ず 1 つだけアクティブアトムを含んでいなければならないということを定めている．ADLMNtal の意味論は基本的には FlatLMNtal に従うが，書換え規則の構文に応じて簡約規則も変えなければならない．FlatLMNtal の簡約規則 [18] を以下に示す．

$$
\begin{array}{lcl}
A & ::= & p(\mathcal{X}) \\
D & ::= & 0 \mid p(\mathcal{X}) \mid D, D \\
P & ::= & A \mid D \mid P, P \mid A, D \text{ :- } P
\end{array}
$$

**図 4　ADLMNtal の構文**

$$
\begin{array}{ccc}
\mathrm{tr}(T, (T \text{ :- } U)) & \xrightarrow{(R6')} & \mathrm{tr}(U, (T \text{ :- } U)) \\
\downarrow & & \downarrow \\
\mathrm{tr}(T), \mathrm{tr}(T \text{ :- } U) & \xrightarrow[*]{(R6'')} & \mathrm{tr}(U), \mathrm{tr}(T \text{ :- } U)
\end{array}
$$

**図 5　変換関数 tr と簡約規則の関係**

$$
\begin{array}{lcl}
\mathrm{tr}(p(\mathcal{X})) & \stackrel{\mathrm{def}}{=} & p_A(L), p_I(\mathcal{X}, L) \\
\mathrm{tr}(P, Q) & \stackrel{\mathrm{def}}{=} & \mathrm{tr}(P), \mathrm{tr}(Q) \\
\mathrm{tr}(T \text{ :- } U) & \stackrel{\mathrm{def}}{=} & \mathrm{select}(\mathrm{tr}(T) \text{ :- } \mathrm{tr}(U))
\end{array}
$$

**図 6　変換関数 tr の定義**

$$
\begin{array}{l}
\mathrm{select}(p_{A1}(L_1), \ldots, p_{An}(L_n), D \text{ :- } P) \stackrel{\mathrm{def}}{=} \\
\quad (p_{Ai}(L_i), D \text{ :- } P, p_{Ai}(L_i), t_1(L_1), \ldots, t_n(L_n)), \\
\quad (p_{A1}(t_1) \text{ :- }), \ldots, (p_{An}(t_n) \text{ :- })
\end{array}
$$

**図 7　変換関数 select の定義**

$$(\mathrm{R6}')\quad T, (T \text{ :- } U) \longrightarrow U, (T \text{ :- } U).$$

ADLMNtal では以上の規則は次の規則に対応する．

$$(\mathrm{R6}'')\quad A, D, (A, D \text{ :- } P) \longrightarrow P, (A, D \text{ :- } P).$$

### 4.2　ADLMNtal の表現力

ADLMNtal では書換え規則の構文に制限が入っているために表現力は FlatLMNtal より低いように見える．しかし，FlatLMNtal を機械的に変換することで FlatLMNtal での計算を再現する ADLMNtal プログラムを得ることができる．つまり，図 5 を可換にする変換関数 tr が存在する．

tr は構文規則について帰納的に定義される．tr はあるアトム $p$ を 2 つのアトム $p_A$ と $p_I$ に分割する．ADLMNtal では $p_A$ はアクティブアトム，$p_I$ はデータアトムである．ここで導入される関数 select は書換え規則の左辺に含まれるアクティブアトムのうち 1 つを恣意的に選択し，またアクティブアトムを削除するような操作を加える．図 7 の定義では $n$ 個のアクティブアトムの中から 1 つだけを選び（それを $p_{Ai}$ とする）それ以外のアクティブアトムを左辺から削除する．この選び方は処理系や引数によって変えても良く，また常に先頭を取るといった方法でも良い．select で右辺に導入されるアトム $t_1, \ldots, t_n$ は一意なシンボルを持つデータアトムである．tr の定義からアクティブアトム $p_A$ を含む書換え規則の左辺は常に対応するデータアトム $p_I$ を含む．したがって，それらの書換え規則と select で新たに作られる書換え規則は危険対を作らないため，書換えの合流性が保たれる．

$\mathrm{tr}(T) = p_{A1}(L_1), \ldots, p_{An}(L_n), D$ とし，$\mathrm{tr}(U) = P$ とおくと，

$$
\begin{array}{ll}
 & \mathrm{tr}(T), \mathrm{tr}(T \text{ :- } U) \\
= & \mathrm{tr}(T), \mathrm{select}(\mathrm{tr}(T) \text{ :- } \mathrm{tr}(U)) \\
= & p_{A1}(L_1), \ldots, p_{An}(L_n), D, \\
 & \quad (p_{Ai}(L_i), D \text{ :- } P, p_{Ai}(L_i), t_1(L_1), \ldots, t_n(L_n)), \\
 & \quad (p_{A1}(t_1) \text{ :- }), \ldots, (p_{An}(t_n) \text{ :- }) \\
\rightarrow & p_{A1}(L_1), \ldots, p_{An}(L_n), P, t_1(L_1), \ldots, t_n(L_n) \\
 & \quad (p_{Ai}(L_i), D \text{ :- } P, p_{Ai}(L_i), t_1(L_1), \ldots, t_n(L_n)), \\
 & \quad (p_{A1}(t_1) \text{ :- }), \ldots, (p_{An}(t_n) \text{ :- }) \\
\rightarrow^* & P, \\
 & \quad (p_{Ai}(L_i), D \text{ :- } P, p_{Ai}(L_i), t_1(L_1), \ldots, t_n(L_n)), \\
 & \quad (p_{A1}(t_1) \text{ :- }), \ldots, (p_{An}(t_n) \text{ :- }) \\
= & \mathrm{tr}(U), \mathrm{tr}(T \text{ :- } U)
\end{array}
$$

であるから，(R6″) の有限回の適用によって (R6′) を再現することができる．したがって，任意の LMNtal プログラムについて等価な計算を行う ADLMNtal プログラムが存在する．

select 関数は複数のアクティブアトムからマッチングに使用するものを 1 つ選択する．このときの選び方によって変換後の ADLMNtal のプログラムの効率は異なる．たとえば，特に個数の多いアトムを選択してしまうとマッチング候補が多くできてしまうため効

率が下がる．FlatLMNtal から ADLMNtal への自動変換を行う場合はより高効率な選び方をすることが重要である．このようなツールは本論文では扱わず，ADLMNtal を記述するユーザが明確にアクティブアトム，データアトムの区別を与えるものとする．

## 5 変換アルゴリズム

この節では 4 節で与えた ADLMNtal のプログラムを C プログラムへと変換する手法を説明する．直感的には，アクティブアトムは C における関数に，データアトムはメモリ上のオブジェクトに対応する．

### 5.1 ADLMNtal の C でのデータ表現

LMNtal の実行環境におけるプロセスの形式化については文献[22] が詳しい．LMNtal アトムは引数に順番を持つため，頂点同士の対によって辺を表現することができない．そこで，アトムと引数番号の組をポート（port）と定義し，ポート同士が互いに参照しているときのみリンクが存在するとする．本論文では ADLMNtal のデータアトムは C でのメモリ上のオブジェクトに対応させる．このとき，データアトムはアトムのラベルを区別するためのフィールドと引数の数だけポートを持つ構造体で表される．また，ポートはあるメモリオブジェクトに対する参照とそのオブジェクトのフィールドをさす参照からなる．たとえば，リストを構成する 2 種類のデータアトム `R = cons(A, B)` と `R = nil` は次の C 構造体定義と対応する．

```
struct port { void *obj; void *ref; };

struct cons {
  int label;
  port A, B, R;
};

struct nil {
  int label;
  port R;
};
```

### 5.2 書換え規則の変換

ADLMNtal の書換え規則の定義から，書換え規則には必ず 1 つアクティブアトムが含まれる．逆に，書換え規則の適用はアクティブアトムが主体となって行われると考えることもできる．その観点では，アクティブアトムは書換え規則の左辺のデータアトムを入力として右辺のプロセスを出力する．このような観察から，本論文ではアクティブアトムは C プログラムにおける関数と対応させる．その定義はアクティブアトムを含む書換え規則から生成される．書換え規則が $A, D$:-$P$ であるとき，生成されるプログラムの疑似コードは次のようになる．

```
void A(...) {
  if (入力が D とマッチングするか) {
    P を生成する;
    D を削除する;
  }
}
```

書換え規則が複数ある場合はそれらの書換え規則による定義が単に並べられる．このとき，並び方によって変換後の結果が異なることがある．LMNtal では書換え規則を適用する順序は非決定的であるため，これに順序を入れると本来得られる複数の実行経路のうち 1 つしか得られない．ただし，変換後のプログラムに対して擬似乱数を用いて条件分岐するコードを書くことで非決定的なプログラムのランダム実行を再現することができる．

### 5.3 変換可能なプログラムの性質

プロセスに複数のアクティブアトムが存在するとき，これらを実行する順序には注意が必要である．上記の方法で変換した ADLMNtal プログラムを C の関数呼び出しによって実行することを考える．C の関数呼び出しは呼び出しスタックを用いて実行されるため，その実行は特定の順序で行われなければならない．具体的には，ADLMNtal プロセスが C の関数呼び出しによって実行されるためには，アクティブアトムは次のような手続きによって実行できなければならない．

1. 初期プロセスのアクティブアトムを好きな順番

でスタックに積む
2. スタックの一番上のアクティブアトムを含む書換え規則を適用する
3. 2 によって新しく生成されたアクティブアトムを好きな順番でスタックに積む
4. 2, 3 をスタックが空になるまで繰り返す

この手続きによって実行可能な ADLMNtal のプロセスは逐次性を持つという．

逐次性を持たないプロセスを実行するためにはある関数の適用を一時中断して他の関数を適用し，後で再開する機能が必要である．C プログラムでこれを実現するためにはスケジューラを実装すればよい．スケジューラは実行していないアクティブアトムを保持しておき，その中から 1 つを選んで実行を試みる．選んだアクティブアトムが書換えに成功すれば新たにできたアクティブアトムをスケジューラに登録する．もし書換えに失敗すればスケジューラは他の実行されていないアクティブアトムの実行を試みる．この手順を繰り返し，アクティブアトムが 0 個になるかすべてのアクティブアトムの書換えに失敗すればスケジューラは停止する．この手続きによって実行可能な ADLMNtal のプロセスは並行性を持つという．

並行性を持つプロセスを実行するその他の方法として，C に提供されているマルチスレッドライブラリなどを用いて OS にスケジューリングを一任するというものが考えられる．この場合，並列にプロセスを書き換えるためには排他制御を行わなければならない．データ構造を並列に操作するアルゴリズムには参照に中間データを用いる手法 [19] やデータにタグを付与する手法 [7] があり，これらを用いることでプロセスを並列に書き換えることができる．

## 6 実装

以上の議論に基づいて，ADLMNtal のプログラムを C へと変換するツール（L2C）のプロトタイプを実装した．L2C には以下のような制約が存在する．

- 入力の ADLMNtal プログラムは逐次性を持つ
- 書換え規則の左辺は連結でなければならない
- ユーザはデータアトム定義，アクティブアトム定義を与える

L2C は LMNtal で実装され，構文解析器などを含め約 800 本の書換え規則からなる．そのうち，提案手法を実装している部分は約 470 本で，残りは構文解析器である．

### 6.1 2 種類のアトムの分類

ADLMNtal には 2 種類のアトムが存在するため，ユーザがそれぞれのアトムを定義する方法が必要になる．

#### 6.1.1 データアトム定義

ここで，データアトムが構成するプロセスの構造を定義する．LMNtal プロセスの構造を型として定める手法については文献 [23] [21] で議論されている．ここではプロセスの集合を生成文法により定め，それを型として扱う．

データアトム定義は `type(`$t$`(`$\mathcal{S}$`)) = {`*Ruleset*`}` のように書く．ここで，$t$ は定義したい型の識別子，$\mathcal{S}$ は $t$ 型のプロセスが持つルートのリンク列を表す．*Ruleset* はプロセス型の構造を生成する規則であり，通常の LMNtal の構文に従うが，左辺は $\mathcal{S}$ を引数に含むアトム 1 つからなり，右辺は任意個の型識別子を名前にもつアトムからなる．*Ruleset* に含まれる規則はアトムを終端記号，リンクを非終端記号とする生成規則である．この観点からは，$\mathcal{S}$ は開始記号とみなされる．右辺の型識別子の引数に与えられるリンクはその型の開始記号と同一視されて新たに生成規則が適用される．

例えば，双方向リストは次のように定義される．

```
type(list(L)) = {
  nil(L) :- .
  cons(A, B, L) :- list(B).
}
```

この規則は，

{L=nil, L=cons(_, nil),
L=cons(_, cons(_, nil)), ...}

というプロセスの集合を与える．右辺に制約のないリンクは任意のプロセスを許す．

データアトム定義は現状の実装では C プログラムにおいて構造体定義を生成するために使われている．6.3 節で述べるように，将来的には最適化に利用する

ことが期待される．

#### 6.1.2 アクティブアトム定義

アクティブアトムはそのアクティブアトムを左辺に含む書換え規則を用いて C 関数定義に変換される．ここでは解析を簡単にするためにあるアクティブアトムについての書換え規則をまとめて定義する構文を与える．

アクティブアトム定義は function($f(\mathcal{A})$) = {*Ruleset*} のように書く．ここで，$f$ は定義したい関数の識別子，$\mathcal{A}$ はアクティブアトム $f$ の引数のリンク列を表す．*Ruleset* は通常の ADLMNtal の書換え規則からなる．ただし，それぞれの規則の左辺は連結でなければならない．

非連結成分を操作するためには，アクティブアトムに直接つながっていないデータアトムを取得する方法が必要になる．そのためには全てのデータアトムを一括で管理するストアを持つというものが考えられる．これはサイズの異なるメモリオブジェクトを効率的に管理，検索，列挙できなければならず，大掛かりな仕組が必要となる．特殊な場合として，0 引数のデータアトムは大域変数をカウンタとして用いることで実装することも可能である．しかしながら，本論文では ADLMNtal を C へ変換する手法の概念実証のためのツールを実装することを目指したため，非連結成分については見送ることとした．

### 6.2 例題

L2C のプロトタイプを用いることで上述の制限を満たす ADLMNtal プログラムから C プログラムを生成することができる．動作を確認した問題にはリスト，木の操作，グラフ構造を持つデータの操作，文脈自由文法による文字列の生成があるが，本論文ではその中から 2 つの例題について述べる．

#### 6.2.1 スキップリストの挿入，検索

スキップリスト [12] は通常の線形リストに加えてノード間を飛び越えるような参照を持つリストである．通常のスキップリストの高さは確率的に決定されるが，ここでは高さが高々 2 で要素の挿入順に依存するとする．このとき，スキップリストのデータ構造は 3 種類のアトムで表すことができる．

```
type(skiplist(P2, P1)) = {
  nil(P2, P1) :- .
  lnode(V, N1, P1) :-
      int(V), skiplist(P2, N1).
  enode(V, N2, N1, P2, P1) :-
      int(V), skiplist(N2, N1).
}.
```

**図 8　スキップリストのデータアトム定義**

図 8 の定義ではスキップリストは 2 つのルートを持ち，高さ 1 のノードを lnode，高さ 2 のノードを enode とする nil で終端された直鎖状のデータ構造になる．スキップリストの挿入，検索操作の定義を図 9 に示す．アトム名の末尾に_l，_e がついているものはそれぞれ高さ 1，2 のノードを走査していることを表している．たとえば insert では，まず高さ 2 のノードを辿り（insert_e），挿入する値より大きいノードの手前から高さ 1 のノードを走査し始める（insert_l）．insert_l は挿入する値より大きいノードか末尾のノードの手前に値を挿入し，スキップリスト全体を昇順に保つ．

L2C が生成した C プログラムは挿入関数が 293 行，検索関数が 374 行であった．

#### 6.2.2 無向グラフの複製

次に，より複雑なデータ構造への操作の例として無向グラフの複製を扱う．無向グラフはノード間に双方向ポインタによってエッジが張られているデータ構造である．LMNtal プロセスのリンクは向きを持たないため，(辺に順序を持つ）無向グラフを自然に表現できる．ここでは簡単のために各ノードの次数は高々3 であるとする．図 10 に無向グラフのデータアトム定義と，複製関数のインターフェースとなるデータアトムの定義を示す．ここでのアクティブアトムは常にインターフェースとなるデータアトムを介して入力を受け取る．

図 11 は複製関数の定義である．複製関数は 3 引数のアクティブアトム copy で表され，第 1 引数のノードを複製して第 2，第 3 引数にそれぞれ返す．このとき，第 1 引数のノードはデータアトム e を通じて受け取る．これは無向グラフが閉路を持つときに copy 同士が衝突したことを検知するためである．copy 同士

```
function(insert_l(A, B, C, D, E)) = {
  insert_l(X, N2, N1, P2, P1), nil(N2, N1) :- int(X) | lnode(X, L, P1), nil(P2, L).
  insert_l(X, N2, N1, P2, P1), lnode(V, NN1, N1) :- X > V  | lnode(V, L, P1), insert_l(X, N2, NN1, P2, L).
  insert_l(X, N2, N1, P2, P1), lnode(V, NN1, N1) :- X =< V | enode(X, N2, L, P2, P1), lnode(V, NN1, L).
  insert_l(X, N2, N1, P2, P1), enode(V, NN2, NN1, N2, N1) :- int(X), int(V) |
      lnode(X, L, P1), enode(V, NN2, NN1, P2, L).
}.
function(insert_e(X, N2, N1, P2, P1)) = {
  insert_e(X, N2, NN1, P2, P1), nil(N2, PP1) :- int(X) | insert_l(X, N2, NN1, P2, P1), nil(N2, PP1).
  insert_e(X, N2, N1, P2, P1), enode(V, NN2, NN1, N2, PP1) :- X > V |
      N1 = P1, enode(V, L2, L1, P2, PP1), insert_e(X, NN2, NN1, L2, L1).
  insert_e(X, N2, N1, P2, P1), enode(V, NN2, NN1, N2, PP1) :- X =< V |
      insert_l(X, N2, N1, P2, P1), enode(V, NN2, NN1, N2, PP1).
}.
function(insert(X, N2, N1, P2, P1)) = {
  insert(X, N2, N1, P2, P1) :- int(X) | insert_e(X, N2, N1, P2, P1).
}.

function(member_l(X, N1, P1, R)) = {
  R = member_l(X, L1, P1), nil(L1, P2) :- int(X) | R = false, nil(P1, P2).
  R = member_l(X, L1, P1), lnode(Y, N1, L1) :- X =:= Y | R = true, lnode(Y, N1, P1).
  R = member_l(X, L1, P1), lnode(Y, N1, L1) :- X < Y | R = false, lnode(Y, N1, P1).
  R = member_l(X, L1, P1), lnode(Y, N1, L1) :- X > Y | lnode(Y, L1, P1), R = member_l(X, N1, L1).
  R = member_l(X, L1, P1), enode(Y, NN2, NN1, P2, L1) :- int(X), int(Y) | R = false, enode(Y, NN2, NN1, P2, P1).
  R = member_l(X, L1, P1), nil(P2, L1) :- int(X) | R = false, nil(P2, P1).
}.
function(member_e(X, N2, N1, P2, P1, R)) = {
  R = member_e(X, L2, N1, P2, P1), nil(L2, NN1) :- int(X) | R = member_l(X, N1, P1), nil(P2, NN1).
  R = member_e(X, L2, N1, P2, P1), enode(Y, NN2, NN1, L2, PP1) :- X =:= Y |
    R = true, N1 = P1, enode(Y, NN2, NN1, P2, PP1).
  R = member_e(X, L2, N1, P2, P1), enode(Y, NN2, NN1, L2, PP1) :- X < Y |
    R = member_l(X, N1, P1), enode(Y, NN2, NN1, P2, PP1).
  R = member_e(X, L2, N1, P2, P1), enode(Y, NN2, NN1, L2, PP1) :- X > Y |
    N1 = P1, enode(Y, L2, L1, P2, PP1), R = member_e(X, NN2, NN1, L2, L1).
}.
function(member(X, N2, N1, P2, P1, R)) = {
  R = member(X, N2, N1, P2, P1) :- int(X) | R = member_e(X, N2, N1, P2, P1).
}.
```

**図 9　スキップリストの挿入，検索**

が衝突した際は b というデータアトムを相手の copy に送る．b を受け取った copy はもう複製するノードがないと判断し終了する．

L2C によって生成された C プログラムの複製関数は 319 行であった．

### 6.2.3　性能評価

LMNtal 処理系 SLIM と L2C が生成した C プログラムとの実行時間を計測した結果を表 1 に示す．L2C が生成した C プログラムは -O オプションをつけてコンパイルした．これにより，アクティブアトムの末尾呼び出しが最適化されることが期待される．

表 1 において，graph_random はランダムな無向グラフ，graph_straight は n2 と n1 のみからなる直線状の無向グラフを入力として無向グラフを複製するプログラムに与えたものである．また，bin_tree は

```
type(graph(X)) = {
  n1(X) :- .
  n2(A, X) :- .
  n3(A, B, X) :- .
}.

type(copy_arg(X)) = {
  e(A, X) :- .
  b(A, B, X) :- .
}.
```

**図 10　無向グラフと複製関数のインターフェースの定義**

2 分木に対して挿入と検索を行うプログラムである．skiplist と bin_tree にはランダムな数値データを与えて挿入，検索した．

SLIM と L2C の性能比に関しては問題によって様々

```
function(copy(Graph, Ret1, Ret2)) = {
  copy(e(X), Ra, Rb), n1(X) :- n1(Ra), n1(Rb).

  copy(e(X), Ra, Rb), n2(A, X) :- n2(A1, Ra), n2(A2, Rb), copy(e(A), A1, A2).
  copy(e(X), Ra, Rb), n2(X, A) :- n2(Ra, A1), n2(Rb, A2), copy(e(A), A1, A2).

  copy(e(X), Ra, Rb), n3(A, B, X) :- n3(A1, B1, Ra), n3(A2, B2, Rb), copy(e(A), A1, A2), copy(e(B), B1, B2).
  copy(e(X), Ra, Rb), n3(X, A, B) :- n3(Ra, A1, B1), n3(Rb, A2, B2), copy(e(A), A1, A2), copy(e(B), B1, B2).
  copy(e(X), Ra, Rb), n3(B, X, A) :- n3(B1, Ra, A1), n3(B2, Rb, A2), copy(e(A), A1, A2), copy(e(B), B1, B2).

  copy(e(L), B, C), e(X, L) :- b(B, C, X).
  copy(e(L), B, C), e(L, X) :- b(B, C, X).
  copy(X, B, C), b(P, Q, X) :- B = P, C = Q.
}.
```

**図 11　無向グラフの複製関数の定義**

**表 1　L2C と SLIM の性能比較 (実行時間 [s])**

| | SLIM | L2C | L2C(tcmalloc) | SLIM/L2C |
|---|---|---|---|---|
| skiplist | 78.056 | 38.482 | 18.304 | 4.26 |
| graph_random | 20.525 | 1.183 | 0.580 | 35.38 |
| graph_straight | 33.038 | 3.476 | 1.793 | 18.42 |
| bin_tree | 52.713 | 24.657 | 10.198 | 5.16 |

である．特に性能比が大きい graph_random については，SLIM が書換え規則ごとにアトムを探索するのに対し，L2C ではアクティブアトムごとに属する書換え規則を試すという処理の順番の違いによる．graph_random では `copy` アトムが多数発生しそれらを書き換えるルールが必ず 1 つ存在するが，SLIM ではそのルールを選ぶまで `copy` アトムを対象として違うルールの適用を試してしまう．L2C により変換された後の C プログラムではそのようなことが起きないため，特に高速になる．skiplist，bin_tree はあまり性能の改善が見られない．これは SLIM が整数を表すアトムについて参照に直接埋め込むという最適化をしており，L2C ではしていないことが一因であると考えられる．

### 6.3　最適化

ADLMNtal から機械的に生成された C プログラムは FlatLMNtal インタプリタより効率的である．しかし，依然として余分なメモリを使用しているところや余計に処理を行っているところがある．それらを削減するためにいくつかの最適化が考えられる．

#### 6.3.1　ポート削減

5.1 節では変換された C プログラムではアトム間の参照が 2 つのフィールドから成るとした．1 つは接続先アトムへの参照で，これはマッチングの際にアトムが条件に合致するかどうかを判定するために使われる．もう 1 つは接続先アトムのどのフィールドに接続しているかを表す．これは書換えの際に接続先アトムのフィールドを書き換えて参照を張り直すために使われる．しかし，データアトム定義ではデータアトムの種類によって接続先のフィールドが決定されることがある．この場合には接続先フィールドの情報は型定義から得られるため実際のデータから削減することができる．

6.2 節のスキップリストの例では，各データアトムのルート以外の引数は接続先のフィールドを静的に決定できる．これらのフィールドについては接続先アトムの情報のみで十分である．また，`insert` と `member` ではスキップリストのルートを張り直しているがルートに接続しているアトムの情報は利用していないため，ルートの接続先フィールドの情報のみで十分である．スキップリストを構成するデータアトムは最大 5 引数であるから，C ではタグフィールドと 5 つのポー

トで 11 ワードの構造体として表される．この最適化により，各ポートは 1 つのポインタとなり 6 ワードの構造体となる．

一方，無向グラフの例ではいずれのデータアトムもどのフィールドに接続しているか決定できない．無向グラフの複製では重複を防ぐために直前に複製したデータアトムがどの引数とつながっていたか確認しなければならない．そのため，接続先フィールドの情報が実行時に必要になる．

#### 6.3.2 アトム，ポート再利用

L2C が生成した C プログラムでは，データアトムの生成，削除に malloc/free 関数を使用している．データアトムは書換えが起きるたびに都度生成，削除されるため，malloc/free 関数が多数呼び出されることとなる．malloc/free 関数が性能に与える影響を確認するために，標準の C アロケータの代わりにより効率の良いアロケータである tcmalloc[14] を使って 6.2 節の例題を計測した．

表 1 から，malloc/free 関数の呼出が性能に大きく影響していることがわかる．ADLMNtal ではデータアトムの種類によってメモリオブジェクトのサイズが決定されるため，書換え規則の両辺に含まれる同種のアトムを確保，解放せずに再利用することで malloc/free 関数の呼び出し回数を減らすことができる．また，そのとき書換えの前後でリンクの接続先が変わらない場合は参照をつなぎかえる回数も減らすことができる．

6.2 節の例では，スキップリストの `member` はデータアトムの構造を変えないため，効果的な最適化が可能である．`member_1` の 4 つ目の書換え規則では，整数アトム 2 個，データアトム 1 個，アクティブアトムのポート 4 個で合わせて 7 回ずつの malloc/free が呼ばれ，リンクの接続が 5 回行われる．最適化をすると malloc/free がなくなり，リンクの接続は 3 回になる．

無向グラフではスキップリストほどの最適化は見られない．`copy` の 2 つ目の書換え規則では，5 回の free と 6 回の malloc，6 回のリンクの接続が行われるが，最適化後は 1 回の malloc，6 回のリンクの接続になる．

#### 6.3.3 参照の単方向化

ADLMNtal から生成される C プログラムではすべてのデータは双方向に参照される．双方向参照はデータの柔軟な操作を可能にするが，参照が単方向にしか利用されない場合は無駄なメモリを使用していることになる．ADLMNtal においてデータアトムの参照を利用するのは以下の 2 通りである．(1) データアトムの引数が局所リンクで，そのリンクを通じてアクティブアトムに到達できないとき．(2) データアトムの引数が自由リンクで，そのリンクをつなぎかえるとき．ただし，(2) はアトムを再利用しかつ余計なリンクのつなぎ替えを行わないという前提がある．

6.2 節のスキップリストの例では，`insert_1` においては `skiplist` のルートの 1 つである `P1` は局所リンクであり (1) に該当しない．`insert_e` においては自由リンクでありつなぎ替えを行わないため，(2) には該当しない．このように見ていくと，`skiplist` の 2 つのルート `P1`，`P2` はどの書換え規則においても上記 2 条件に該当しないことがわかる．したがって，このプログラムにおいては逆向き参照である `P1`，`P2` は必須ではなく，単方向スキップリストで十分であるということがわかる．ポート削減と組み合わせれば，スキップリストの 1 つのノードは 4 ワード（値，ノードの種類，次の要素への 2 つの参照）で表される．

無向グラフの例ではどの参照も双方向であることを意図しているが，この最適化を適用しても参照が単方向になることはない．

## 7 まとめと今後の課題

本論文では，グラフ書換え言語である LMNtal から C プログラムを機械的に生成するための手法を提案した．LMNtal と C の間に存在する大きな隔たりを解消するため，LMNtal に制約を加えた ADLMNtal を定義した．ADLMNtal で記述された例題から C プログラムが生成でき，元の LMNtal 処理系よりも効率的に動くことを確認した．今後の課題としては以下のものが考えられる．

5 節では ADLMNtal プログラムの逐次性，並行性を導入した．6 節では逐次性を持つ ADLMNtal プログラムを対象としており，与えられた ADLMNtal プ

ログラムが逐次性を持つかどうかを判定することは重要である．逐次性判定はデータアトムに対して構造型と入出力型を導入することで可能だと考えられる．

6 節では非連結成分を扱うための手法について述べた．しかし，5 節で述べたように並列な C プログラムを生成する場合は，大域的な領域に共有資源があると問題になりやすい．これらの手法を効率よく実装することは，実用上重要である．

**謝辞** 本研究を進めるにあたって活発な議論をしてくださった上田研究室の皆様と，松本翔太氏，恒川雄太郎氏に感謝いたします．

## 参 考 文 献

[1] Agrawal, A., Karsai, G., and Lédeczi, Á.: An End-to-end Domain-driven Software Development Framework, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, ACM, 2003, pp. 8–15.

[2] Bak, C. and Plump, D.: Compiling Graph Programs to C, *International Conference on Graph Transformation (ICGT 2016)*, LNCS, Vol. 9761, Springer, 2016, pp. 102–117.

[3] Bakst, A. and Jhala, R.: *Predicate Abstraction for Linked Data Structures*, Springer Berlin Heidelberg, 2016, pp. 65–84.

[4] Booch, G., Jacobson, I., and Rumbaugh, J.: *The unified modeling language reference manual*, 1999.

[5] Karsai, G., Agrawal, A., Shi, F., and Sprinkle, J.: On the use of graph transformations in the formal specification of computer-based systems, *Proceedings of IEEE TC-ECBS and IFIP10*, Vol. 1(2003), pp. 19–27.

[6] Löwe, M. and Beyer, M.: AGG – An implementation of algebraic graph rewriting, *Rewriting Techniques and Applications: 5th International Conference (RTA–93)*, LNCS, Vol. 690, Springer, 1993, pp. 451–456.

[7] Michael, M. M.: High Performance Dynamic Lock-free Hash Tables and List-based Sets, *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, New York, NY, USA, ACM, 2002, pp. 73–82.

[8] OMG: *"MDA – The Architecture Of Choice For A Changing World"*, Needham, MA, 2017. [Online] `http://www.omg.org/mda/`.

[9] OMG: *"OMG's MetaObject Facility"*, Needham, MA, 2017. [Online] `http://www.omg.org/mof/`.

[10] Plump, D.: The graph programming language GP, *International Conference on Algebraic Informatics*, LNCS, Vol. 5725, Springer, 2009, pp. 99–122.

[11] Plump, D.: The Design of GP 2, *10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, EPTCS, Vol. 82, 2012, pp. 1–16.

[12] Pugh, W.: *Skip lists: A probabilistic alternative to balanced trees*, Springer Berlin Heidelberg, 1989, pp. 437–449.

[13] Reynolds, J. C.: Separation logic: A logic for shared mutable data structures, *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, IEEE, 2002, pp. 55–74.

[14] Sanjay, G. and Menage, P.: "TCMalloc : Thread-Caching Malloc", 2017. [Online] `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[15] Schürr, A.: PROGRES, A Visual Language and Environment for PROgramming with Graph REwriting Systems, Technical Report AIB 94–11, RWTH Aachen, Germany, 1994.

[16] Schürr, A., Winter, A. J., and Zündorf, A.: The PROGRES approach: Language and environment, *Handbook of graph grammars and computing by graph transformation*, World Scientific Publishing, 1999, pp. 487–550.

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

[18] Ueda, K. and Kato, N.: LMNtal: a language model with links and membranes, *International Workshop on Membrane Computing (WMC 2004)*, LNCS, Vol. 3365, Springer, 2004, pp. 110–125.

[19] Valois, J. D.: Lock-free linked lists using compare-and-swap, *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, ACM, 1995, pp. 214–222.

[20] Vizhanyo, A., Agrawal, A., and Shi, F.: Towards generation of efficient transformations, *International Conference on Generative Programming and Component Engineering (GPCE 2004)*, LNCS, Vol. 3286, Springer, 2004, pp. 298–316.

[21] 吉元 佑介, 上田 和紀: グラフ書換え系における静的グラフ型検査, 日本ソフトウェア科学会第 *32* 回大会, 2015.

[22] 信夫 裕貴, 田辺 良則, 上田 和紀: LMNtal におけるグラフ書換え操作の Coq による形式化, 日本ソフトウェア科学会第 *30* 回大会, 2013.

[23] 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.