帰納的学習を用いたXMLスキーマの統合に関する基礎的研究

小川 順平 † 粕谷 英人 †† 大久保弘崇 †† 山本 晋一郎 ††

† 愛知県立大学大学院 情報科学研究科 †† 愛知県立大学 情報科学部

Integration of XML Schemata by Induction Learning

Junpei OGAWA† Hideto KASUYA†† Hirotaka OHKUBO†† Shinichiro YAMAMOTO††

†Graduate School of Information Science and Technology, Aichi Prefectural University
††Faculty of Information Science and Technology, Aichi Prefectural University

1 はじめに

各種の CASE ツールが共通に必要とする機能を提供する基盤ソフトウェアである CASE ツールプラットフォームは，汎用のデータ交換フォーマットとして急速に普及しつつある XML を多く採用している．XML を利用した CASE ツールプラットフォームには JX-model[2]，ACML[4] などが挙げられる．

XML は本来汎用のフォーマットであるが，CASE ツールプラットフォームがそれぞれ異なる XML スキーマを採用しているので，CASE ツールは利用した CASE ツールプラットフォームが用いる XML スキーマに依存する．

異なる XML スキーマに基づくプラットフォームの間でデータ交換を行う，あるいは CASE ツールの機能を共有するためには，全ての XML スキーマ間のコンバータを作成するか，または全てのプラットフォームに対して CASE ツールを移植する必要がある．しかし，今後さらに XML スキーマが増えることが予想されるため，これらの選択はいずれも現実的ではない．

このことは CASE ツール分野以外の XML スキーマにも当てはまる．XML スキーマが乱立することで，XML を利用するツールを作成する際には，本質的に同じ情報を表現する多種の XML スキーマからの選択を迫られることになる．またツール作成後に，他のスキーマが主流となることもある．場合によっては主流ではないスキーマのサポートがなくなり，それを対象としていたツールが使用できなくなる可能性もある．

この問題は，XML を利用するツール自体が特定の XML スキーマに必然的に依存してしまうことが原因である．ある XML スキーマに依存したツールが異なる XML スキーマで表現された本質的に同じ情報を利用できるようにできれば，この状況は打開できる．そこで，XML スキーマ間のデータ互換性を高めるために，ある XML スキーマで表現されているデータを他の XML スキーマに自動変換するが重要である．

本研究では，木構造間の変換を自動的に行う方法として，帰納的学習に着目した．帰納的学習を用いて二つの XML スキーマ間の写像変換を行うプログラムを学習させる．

木構造間の写像変換の帰納的学習の例として，S 式の対の写像変換を行う LISP プログラムの学習を挙げる． LISP で扱う S 式の本質は二分木であり，S 式を写像変換を行う LISP プログラム学習は木構造間の変換の学習に適用できる部分があると考えられる．LISP プログラムの帰納的学習の性質をまとめ，学習可能な写像変換を調査する．そして，LISP プログラムの帰納的学習を XML スキーマ間の写像変換に適用する際に考えられる問題点について考察する．帰納的学習の性質を理解した上で，帰納的学習によって学習可能であることと不可能であることを切り分ける．

以下，2 章では，帰納的学習に対するこれまでの取り組みについて木構造の写像学習について LISP を例に挙げて考察する．3 章では，CASE ツールのデータフォーマットとして使われている XML ツールの具体例について述べ，2 章で述べた帰納的学習を XML に適用する際の問題点について考察する．4 章では，まとめと今後の課題を述べる．

2 帰納的学習

帰納的学習とは，与えられた個々の事例から一般的・普遍的な規則を導き出す推論である．本研究での帰納的学習は，“計算機に陽にプログラムを与える以外の方法によって，計算機がプログラムを獲得すること” として扱う．これまで機械学習における帰納的学習は数多く研究されている．そこで，文字列と木構造データを扱う帰納的学習の例を挙げる．

2.1 文字列の学習

ある文字列の集合を与え，それらを受理するような有限オートマトンや文脈自由文法を学習する研究 [7] がある．

有限オートマトンの学習は，与えられた文字列の増え方が右線形だけといった 1 方向への解決を考えれば十分である．すなわち，一つずつ文字を受け取り，受け取る度に状態を遷移するような有限オートマトンを作る．与えられた文字列を受理する有限オートマトンを全て求め，それらの中から最適解を探索する．受理させたい文字列を与えてゆき，それらが全て受理するかを確認する．

文脈自由文法の学習では，文字列に対する全ての可能な文法構造を考える必要がある．例えば，文字列 $w = aabb$ を受理する文脈自由文法を学習させるためには，図 1 のような文法構造全てを考える必要がある．

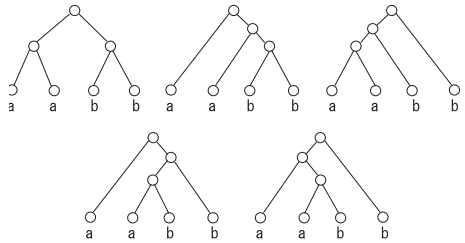

図 1: 文字列 “*aabb*” に対する全ての可能な文法構造

解を探索する候補を全て求めたら，最適解の探索方法は有限オートマトンと同様である．他の受理させたい文字列を候補の全てに一つずつ与えていき，受理しないものを切り捨てていく．そして残ったものを最適解とする．

2.3 節から挙げる木構造データの学習も，候補からの解の探索問題はこれらと同様の方法で考えられている．

2.2 木構造データの学習

本研究では，XML スキーマの対の写像変換が目的である．木構造データの写像変換の参考として，S 式の対を与えその写像を行う LISP プログラムを獲得する帰納的学習を挙げる．S 式はリスト構造をしているので，木構造間の変換の学習に適用できる部分があると考えられる．2.3 節では，LISP プログラムの帰納的学習の中でも，アルゴリズミックな方法で行っている Biermann の regular LISP programs[6] について調査する．

ある S 式の対 (X, Y) の集合を与え，入力 S 式 X から出力 S 式 Y に写像変換を行う LISP プログラムを生成する帰納的学習を Biermann が提案している．

例えば，ある入力 S 式を与えるとそれを再帰的にリバースした S 式を返してくれるプログラムがほしいと考える．そこで入出力対 (X, Y) として入力 $X = ((A\,.\,B)\,.\,C)$ と出力 $Y = (C\,.\,(B\,.\,A))$ という S 式の対を与えると，原始関数の cons, car, cdr のみで X を使って Y を表すと，$Y =$ (cons (car X)(cons (cdr (car X))(car (car X)))) と書くことができる．しかし，この (X, Y) の関係は一つの入出力の対に対しての計算であるため，この計算結果が他の入力に対しても望まれた振る舞いを行うとは限らない．入出力対の集合の全ての要素に対して，望まれた振る舞いを行う LISP プログラムを獲得することが必要である．これを実用的な時間で最小の入出力対情報を用いて可能にするために，Biermann は regular LISP programs を定義した．

2.3 Regular LISP programs

2.3.1 特徴

どんな入出力対の集合に対しても望まれた振る舞いをするプログラムを獲得できるようにするためには，入出力対に対してあらゆる可能性を考える必要がある．あらゆる可能性について考えるとプログラムの探索空間は大きくなり，解を見つける計算時間はそれにともない大きくなってしまう．自動プログラム生成の利用者は，可能な限り直接的な方法で望まれた計算を例を通して表現したい．不必要なあいまいさは採用したくない．そこで，実用的な時間でプログラムを獲得するために，探索するドメインを制限した．

最初に，X 内の全てのアトムが NIL ではなく区別がつくということを必要とした．これは，Y 内のアトムはいつでも X 内の位置が唯一に判断できるということである．例えば，$X = ((A\,.\,B)\,.\,C)$ としたとき $A = (\text{car}\ (\text{car}\ X))$ と書けるが，$X = ((A\,.\,A)\,.\,C)$ としたとき A を一意に表すことはできない．そのため，こういった入力を扱ってはいない．また，X 内にはないアトムは Y 内には現れない．$X = (A\,.\,B)$ としたときに，$Y = (A\,.\,C)$ のような対は扱えない．なぜなら入力 X を用いてアトム C 表すことができないからである．さらに regular LISP programs で扱う関数は原始関数 (car, cdr, cons, atom, cond) に限られている．これらの特徴によって，リスト構造の写像変換を可能にしている．

2.3.2 定義

Regular LISP programs とは semiregular LISP programs に二つの条件を加えたものである．

Semiregular LISP programs とは次の形をした $F_1, \cdots, F_n$ である:

$$\begin{aligned}(F_i\ X) \quad = \quad & (\text{cond}\ (p_{i1}\ f_{i1}) \\ & \qquad (p_{i2}\ f_{i2}) \\ & \qquad\quad \vdots \\ & \qquad (p_{in}\ f_{in}))\end{aligned}$$

ここで，

- $p_{i1}, \ldots, p_{in}$ は以下の条件を満たす述語である:
  - $p_{ij} = (\text{atom}(\text{C}w_{ij}\text{R}\ X))$, ただし $w_{ij} \in (\text{A} + \text{D})^*$, w_{ij} は w_{ij+1} の接頭辞である．
  - $p_{in} = \text{T}$
- $f_{i1}, \cdots, f_{in}$ は次の形式の一つである:
  $\text{nil}, X, (F_h\ (\text{car}\ X)), (F_h\ (\text{cdr}\ X)), (\text{cons}\ (F_h\ X)\,(F_k\ X))$

さらにこの定義に以下の二つの性質を付け加えたものが regular LISP programs である．

- **direct である**
  入力 $X = (A\,.\,B)$ を出力 $Y = (A\,.\,A)$ に変換するプログラムを考える．これは $Y = (\text{cons}\ (\text{car}\ X)\,(\text{car}\ X))$ と書ける．同じ意味を持つ semiregular LISP programs に従うプログラムが 2 つ書ける:

$$\begin{aligned}(F_1\ X) &= (\text{cons}\ (F_2\ X)\,(F_3\ X)) \\ (F_2\ X) &= (F_4\ (\text{car}\ X)) \\ (F_3\ X) &= (F_5\ (\text{car}\ X)) \\ (F_4\ X) &= X \\ (F_5\ X) &= X\end{aligned}$$

もしくは次のようにも書ける:

$$\begin{aligned}(G_1\ X) &= (G_2\ (\text{car}\ X))\\ (G_2\ X) &= (\text{cons}\ (G_3\ X)(G_4\ X))\\ (G_3\ X) &= X\\ (G_4\ X) &= X\end{aligned}$$

Y において car は cons の両方の引数に適用されている．従って，F のように car を cons の後で適用する方法と，G のように前に適用する方法の 2 通りの方法がある．regular LISP programs では常に G のように記述する．このように原始関数の適用に優先度を与えることを direct であるという．

- **car-nil, cdr-nil がない**
  二つめの条件は，“car-nil” と “cdr-nil” を使用しないことである．car-nil の説明のために，次の例を考える:

$$\begin{aligned}(F_1\ X) &= (F_2\ (\text{car}\ X))\\ (F_2\ X) &= \text{nil}\end{aligned}$$

  $(F_1\ X)$ の値が nil であることと (car X) の計算が無駄であることは明らかである．このことを “car-nil” instance という．$(F_1\ X) = \text{nil}$ と定義すれば、car-nil を避けることができる．このように，$(F_1\ X)$ を評価して nil になった場合，その間に無駄な CAR や CDR の計算がないように，原始関数を適用する必要がある．

以上が regular LISP programs の定義である．2.3.3 節では，これらの定義に従った regular LISP programs を生成する帰納的学習のアルゴリズムを示す．

2.3.3 例からの LISP プログラムの生成

実際に，例が与えられてから LISP プログラムを求めるアルゴリズムについて述べる．2.3.1 節で述べた性質を持つ S 式の対 (X, Y) が与えられたとき，帰納的学習のアルゴリズムは，まず “computation trace” $t(X, Y)$ を作成する．

“composition” $c(X, Y)$ が次のように求められるとき，

$$c(X,Y) = \begin{cases} \text{nil},\ Y \text{ が nil のとき} \\ X \text{ を使用して書かれた } Y^1\ , \\ \quad Y \text{ が } X \text{ 内にあるアトムであるとき} \\ (\text{cons}\ c(X,(\text{car}\ Y))\ c(X,(\text{cdr}\ Y))), \\ \quad \text{その他のとき} \end{cases}$$

$t(X, Y)$ の定義は以下の通りである．

$$t(X,Y) = \begin{cases} (\bar{N}),\ \ c(X,Y) = \text{nil のとき} \\ (\bar{I}),\ \ c(X,Y) = X \text{ のとき} \\ (\bar{A}\ t((\text{car}\ X),Y)), \\ \quad (\text{car}\ X) \text{ が } c(X,Y) \text{ 内に現れ，かつ} \\ \quad (\text{cdr}\ X) \text{ と } X \text{ が cons の引数にはない，または} \\ \quad X \text{ が単独で } c(X,Y) \text{ 内に現れる} \\ (\bar{D}\ t((\text{cdr}\ X),Y)), \\ \quad (\text{cdr}\ X) \text{ が } c(X,Y) \text{ 内に現れ，かつ} \\ \quad (\text{car}\ X) \text{ と } X \text{ が cons の引数にはない，または} \\ \quad X \text{ が単独で } c(X,Y) \text{ 内に現れる} \\ (\bar{O}\ t(X,(\text{car}\ Y))\ t(X,(\text{cdr}\ Y))),\ \text{その他のとき} \end{cases}$$

この定義に従って，S 式の入出力対 (X, Y) が $X = ((A\,.\,B)\,.\,C)$，$Y = (C\,.\,(B\,.\,A))$ のように与えられたとき，$t(X, Y)$ は次のようになる．

$$\begin{aligned} t(X,Y) &= (\bar{O}\ t(X,C)\ t(X,(B\,.\,A))) \\ &= (\bar{O}(\bar{D}\ t(C,C))(\bar{A}\ t((A\,.\,B),(B\,.\,A)))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}\ t((A\,.\,B),B)t((A\,.\,B),A)))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}(\bar{D}\ t(B,B))(\bar{A}\ t(A,A))))) \\ &= (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{I}))))) \end{aligned}$$

この computation trace の記号 $(\bar{N}, \bar{I}, \bar{A}, \bar{D}, \bar{O})$ を LISP の演算 (nil, identity, car, cdr, cons) を実行する関数に見立てると，上の例 $t(X, Y) = (\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{O}(\bar{D}(\bar{I}))(\bar{A}(\bar{I})))))$ が図 2 のように表せる．computation trace を計算することで，与えられた入出力対 (X, Y) に対して X から Y を得るための唯一の regular LISP programs を求めることができた．図 2 を見ると有限オートマトンに似ていることがわかる．あとは全ての入出力の対に対して望まれた振る舞いをする有限オートマトンの最小の解を求める問題に帰属させて考える．この例は入力 X のリバース出力 Y を得る LISP プログラムを獲得することが望まれるので，他のリバースをするような入出力対 (X, Y) を与えていき最適な解を探索していく．すると，図 3 のように最小の解を求めることができる．

図 2 から図 3 を求めるためには，根のノードから下の子に向かって，自分が自分の祖先のノードとマージできないか一つずつ判断していく．マージの方法は，まずマージしてみて，与えれた全ての入出力対に対して望まれた振る舞いをすることを確認してエラーがなければその二つのノードをマージする．この例の場合は，$\{F_1, F_4, F_5, F_8, F_9\}, \{F_2, F_6\}, \{F_3, F_7\}$ のようにマージされる．

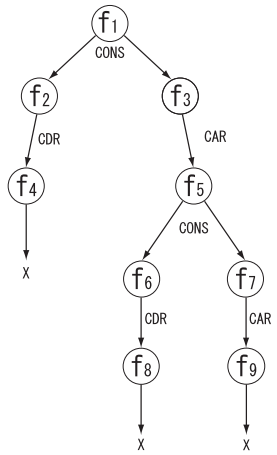

図 2: coputation trace の例

[1] $X = ((A\,.\,B)\,.\,C)$ としたとき $A = (\text{car}\ (\text{car}\ X))$ と書く．

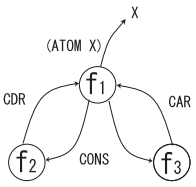

図 3: リバースプログラムの最小解

2.3.4 学習能力

ここまで述べたように，LISP プログラムの探索を実用的な時間に抑えるためや直接的にプログラムを生成を可能にするために，regular LISP programs に次のような性質を持たせていた．

1. 扱う基本関数は原始関数の car, cdr, cons, atom, cond のみである．
2. S 式の入出力対 (X, Y) について，X 内の全てのアトムが NIL ではなくそれぞれ記号が異なる．
3. S 式の入出力対 (X, Y) について，Y 内の全てのアトムは X 内のアトムのいずれかである．

学習可能

これらの性質から，入れ子構造などの構造の写像変換の学習について，regular LISP programs は適切な LISP プログラムを求めることができる．regular LISP program を利用して生成できる LISP プログラムの例を挙げる．例えば，入出力対 (X, Y) として $X = ((A\,.\,B)\,.\,C)$, $Y = (C\,.\,(B\,.\,A))$ のように再帰的にリバースさせている入出力対の集合が与えられたときの LISP プログラムは次のようになる．

$$
\begin{aligned}
(F_1\ X) &= (\text{cond}\ ((\text{atom}\ X)\ X) \\
&\qquad (T\ (F_2\ X)(F_3\ X))), \\
(F_2\ X) &= (F_1\ (\text{cdr}\ X)), \\
(F_3\ X) &= (F_1\ (\text{car}\ X))
\end{aligned}
$$

入力 X に対してそのリバース Y を出力をしてくれる最小の LISP プログラムを学習する．この LISP プログラムは他の全ての入出力対に対しても望まれた振る舞いをしている．

学習不可能

前述の性質により，regular LISP programs を利用して解決できない学習について述べる．項目 1 より，扱える関数にはリストの分解・結合を行う関数のみであり，アトムの記号を評価するような原始関数がない．そのため，与えられたリストのそれぞれのアトムの関係を利用した学習は行えない．あくまでもアトムの記号は，それぞれを識別するためだけにしか扱っていない．項目 2 より，入力 X には同じ記号のアトムを複数使用できないため，同一アトムの数を扱った学習は行えない．項目 3 より，X 内に存在しないアトムを Y 内で扱えないため，Y で新たにアトムを生成させるさせるような学習はできない．また，アトムを生成するような関数も扱っていない．

このように，regular LISP programs で行える帰納的学習は，S 式の入れ子構造の写像変換のみに特化した性質になっているため，学習できる内容が大きく制限されている．

3 XML スキーマの写像変換

2.3 節では，LISP の木構造の写像変換の帰納的学習について述べた．本研究では，ある XML スキーマから他の XML スキーマへの写像変換を目的としている．そこで，LISP の木構造の写像変換を XML スキーマに当てはめることを考える．XML スキーマの例を挙げて，LISP のプログラム合成を適用する上での問題点について考察する．

3.1 XML スキーマ

XML (Extensible Markup Language) は，W3C により 1998 年 2 月に XML1.0 の勧告が公開されたデータ記述用メタ言語である．XML ソースは「要素」(Entity) と「属性」(Atribute) が複数集まって構成されている．要素は内部に子要素を含むことができる．そのため，木構造として扱うことができる．また，属性は要素に付随し，内部に子要素を含むことはできない．

XML を利用した CASE ツールプラットフォームがプログラムソースを解析したデータ保存形式の例を表 1 に示す．同じプログラミング言語を対象にしていても，各スキーマごとにオリジナルソースの保存の観点だけを見ても特徴がある．

例えば，OOML[3] と srcML[5] はともに C++言語を対象としている．srcML はオリジナルソースを完全に保存するが，OOML はまったく保存しない．このように，CASE ツールプラットフォームによって重要としているものが異なることが原因で，同じソースプログラムを解析した結果である XML スキーマであっても構造が異なっている．そのため，ある XML スキーマを扱うツールが他のスキーマを扱えないという現状がある．

表 1: CASE ツールプラットフォーム例

	対象言語	ソース保存
JavaML[1]	Java	×
JX-model[2]	Java	完全
OOML[3]	Java, C++	×
srcML[5]	C++	完全
ACML[4]	ANSI C	空白以外

Java 言語を対象としている JavaML と JX-model の XML スキーマを例として取り上げ，帰納的学習を木構造の自動変換に適用する際の問題点について考察する．この 2 者は，3.2 節で述べるように大きく構造が異なる．よって，この 2 者の相互変換が実現できれば，他の構造がより類似している XML スキーマ間変換はより容易に可能と考えられる．

3.2 JavaML と JX-model

以下にメソッド宣言を表すプログラムソース断片の解析結果である XML ソースを示し，その差異について述べる．JX-model の出力は，構造を見やすくするために編集してある．

プログラムソースのメソッド宣言部分

```
void foobar() {}
```

JavaML

```
<method name="foobar" id="test:mth-13" line="5"
             col="4" end-line="5" end-col="14">
  <type name="void" primitive="true"/>
  <formal-arguments/>
  <block line="5" col="13" end-line="5" end-col="14">
  </block>
</method>
```

JX-model

```
<Method id="s5913968641" typefirst="s5922357249">
  <Type id="s5922357249" sort="Primitive">
    <kw>void</kw>
  </Type>
  <sp> </sp>
  <ident defid="s5913968641">foobar</ident>
  <op>(</op>
  <op>)</op>
  <sp> </sp>
  <Stmt id="s5947523073" sort="BLOCK">
    <op>{</op>
    <op>}</op>
  </Stmt>
</Method>
```

1. JX-model はオリジナルソースとコメントを完全に保存するのでテキストの位置情報やスペースなどの情報を保持しているが，JavaML はオリジナルソースを完全には保存しない．
2. JX-model はテキストノードを持つ．JavaML はソースプログラムの情報を要素と属性の値で全てを表し，テキストノードを持たない．
3. 要素の種類は，JX-model が 27 種，JavaML は 76 種．

項目 1 より，JX-model より JavaML はオリジナルソースのテキストの位置情報について情報が少ないことが分かる．XML の例を見ると，JavaML では `method` 要素の属性にそのメソッドのテキストの開始位置情報を持っているが，インデントなどの詳しいテキスト情報を得ることはできないが，JX-model ではタグを外してしまえば，完全にもとのオリジナルソースに戻すことができる．

項目 2 については，JX-model はタグを外すことによってオリジナルソースに戻せるようにしているため位置情報を持つようになっている．JavaML は要素の属性の値にプログラムソース断片をそのまま文字列として与えたり，ソース内容と同じ意味の表現をさせているものもある．XML の例について述べると，JavaML のメソッド名の表し方は `method` 要素の属性に `name="foobar"` のようにソースのテキストのメソッド名をそのまま値として文字列を与えている．void 型が primitive であるという情報の表し方は，`type` 要素の属性 `primitive="true"`というように，ソースのテキストをそのまま使わずに，そのメソッドが primitive であるかどうかを表している．そのため，2 つのスキーマは本質的に同じ内容を扱っていても構造が大きく異なる場合がある．

項目 3 については，JX-model では要素とその属性の値によってテキストノードの情報を表しているが，JavaML では細かい要素分けにより表現している．

3.3 写像変換の問題点

LISP プログラムの帰納的学習を適用したときの 2 つの XML スキーマの写像変換の問題点について，3.2 節で載せた双方の XML ソースを比較しながら考察する．

- **構造の知識**

  メソッド名情報に着目する．メソッド名 `foobar` を，JavaML では `method` 要素の属性の値に保持している．JX-model は `Method` 要素の子の `ident` 要素で保持している．メソッド名のように同じ文字列を用いて情報を表している場合の写像変換は，その文字列を表している場所 (構造) を変換することになる．構造を写像変換をするには，それぞれのメソッド名を持つ場所の知識を予め与えるなどの学習をさせる必要がある．

- **内容の知識**

  メソッドの型である `void` がプリミティブであるという情報の表し方を見る．JavaML では `type` 要素の属性 `primitive="true"`で表されている．JX-model では `Type` 要素の属性 `sort="Primitive"` で表される．この 2 者の表現が同じ情報であるということを判断しなければならない．

構造の知識について考える．要素の入れ子などの構造の写像変換に関しては Biermann の手法を利用することができるが，要素・属性の名前は唯一でなければならないという条件が必要である．XML を扱う場合，同じ名前の要素・属性を複数扱うことは通常よくあることなので，これに対応した写像変換が不可欠になる．さらに，属性の値で保持している情報を要素のテキストノードで表すといった変換も必要になる．この変換は Biermann の手法を用いることはできない．なぜなら Biermann の手法はリストの入れ子構造を解決することはできても，アトムの文字列を全体・もしくは部分的に扱うことはできないからである．すなわち，`method` 要素をより下の子の要素と入れ替えたりはできるが，`method` 要素の `name` 属性が持つ `foobar` というメソッド名を `foobar` という文字列を取り出して扱うことはできないのである．なぜなら，Biermann の手法では，`name="foobar"`の文字列全体をアトムとして扱い，その部分文字列 `foobar` を取り出すためには文字列を扱う関数が必要になるからである．

内容の知識については，Biermann の手法を利用することはできない．この手法は，リスト構造の問題についてしか言及していないからである．前述の `primitive="true"`と `sort="Primitive"` この 2 者の表現が同じ情報であるということを判断して，それらを写像変換することが必要になる．そのためには，どの要素にどんな情報を持っているのか，そのスキーマではその情報をどのように表すのかといったことを，これらを例として与え帰納的学習で獲得できるようになる必要がある．

このように XML の写像変換を行うためには，構造だけではなく文字列を扱う写像変換が必要になる．文字列を扱うためには，Biermann の手法のリスト構造を扱う基本関数 (CAR, CDR,

ATOM, CONS) のみでは不可能である．これらの関数はリストからアトムを取り出したり，リストの分解・結合しか行うことしかできない．また，アトムの文字列を評価する関数が少なくとも必要である．Biermann の手法では，アトムを評価する関数がアトムであるかを判別する atom しかない．アトムの文字列を比較したり，文字列を扱う関数が必要である．

アトムの文字列比較が必要な帰納的学習の例を挙げる．例えば，入出力対の集合

$$\begin{aligned}(A.(A\ B\ C)) &\rightarrow B\\ (B.(A\ B\ C)) &\rightarrow C\\ (C.(A\ B\ C)) &\rightarrow \text{nil}\end{aligned}$$

から,「(cdr X) から (car X) と等しい要素を探し，その次の要素を返す」プログラムを学習するには，アトムの等価性判定を行う関数が不可欠である．

また，JavaML では `method` 要素、JX-model では `Method` 要素はともにメソッド宣言の内容を表しているが，写像変換するためには要素名を変換しなければならない．他方のスキーマにはない要素などがあるときもある．従って，文字列を変換もしくは生成する関数も必要である．

以上のように，XML の写像変換において，構造の知識によるものはこれまでの LISP プログラム自動合成の技術を適用することによって木の構造のみを写像変換することは学習可能である．しかし，文字列や内容の知識についての写像変換を行うためには，LISP プログラム自動合成で扱う関数を増やす必要がある．また，どの要素にどんな情報を持っているのか，そのスキーマではその情報をどのように表すのかといった知識を獲得するために，予め知識を与えるまたは帰納的学習をする際に獲得できるようにするということが考えられる．

4 おわりに

4.1 まとめ

本論文では，XML スキーマの自動変換を帰納的学習を用いて行う目的のために，帰納的学習のこれまでの取り組みについてまとめた．そして，XML スキーマの写像変換の帰納的学習をリスト構造の LISP プログラム合成を適用したときの問題点について考察した．

XML の写像変換において，構造の入れ子に関する変換はこれまでの LISP プログラム自動合成の技術を適用することが可能であることがわかった．文字列や内容の知識についての写像変換を行うためには，LISP プログラム自動合成で扱う関数を増やすなどの拡張が必要である．

4.2 今後の課題

今後の課題を，以下に挙げる．

- **基本関数に必要な関数の把握**

  LISP プログラムの帰納的学習を適用する際に，XML の入れ子構造を扱うのに加えて，文字列情報と内容の知識を扱うために必要な基本関数について改めて詳しく把握する必要がある．LISP の原始関数として CAR, CDR, ATOM, CONS, COND, QUOTE, EQ が挙げられる．これらを基本関数として，Biermann の手法を見直す．ただし，基本関数が増えることによって，プログラム生成の探索空間が大きくなってしまうので，最適な解を求めるための最小の基本関数の集合を見つける必要がある．

- **LISP プログラム自動合成の拡張**

  基本関数が増えることによって，LISP プログラム自動合成のアルゴリズムもそれにともなって拡張する必要がある．regular LISP programs に新しい基本関数を応用させ，再定義を行う．

- **XSLT への応用**

  今回参考にした LISP プログラムの帰納的学習は S 式の写像変換のために考えられている．本研究の対象は XML であり，その性質に合わせた変換ができると効率がよい．今日 XML を扱うのに用いられている XSLT を帰納的学習に適用することができたら，より XML に特化した効率のよいアルゴリズムが提供できると考えられる．

参考文献

[1] Greg J. Badros : “JavaML:A Markup Language for Java Source Code”, Proceedings of 9th International World Wide Web Conference (WWW9), Amsterdam, The Netherlands, 13-15 (2000/5)

[2] 吉田 一, 山本 晋一郎, 阿草 清滋 : “XML を用いた汎用的な細粒度ソフトウェアリポジトリの実装”, 情報処理学会 OO2002 シンポジウム, pp.83–90 (2002/8)

[3] E. Mamas and K. Kontogiannis : “Portable Source Code Representation Using XML”, Proceedings of the Seventh Working Conference on Reverse Engineering, IEEE Computer Society Press, Brisbane Australia, 23–25, pp. 172–182.(2000/11)

[4] 川島勇人, 権藤克彦 : “XML を用いた ANSI C のための CASE ツールプラットフォーム”, 情報処理学会 OO2003 シンポジウム, pp.187–188 (2003/8)

[5] Collard, M.L., Kagdi, H., Maletic, J.I. : “An XML-Based Lightweight C++ Fact Extractor”, Proceedings of the IEEE International Workshop on Program Comprehension (IWPC03), Portland, OR, pp. 124–143(2003)

[6] Biermann, A. W. : “The Inference of Regurlar LISP Programs from Examples”, Duke Univ., IEEE Transactions on Systems, MAN, and Cybernetics, Vol. SMC-8, NO.8, pp.585–600 (1978/8)

[7] Biermann, A. W. and Feldman, J. A. : “On the Synthesis of Finite-State Acceptors”, A. I. Memo No. AIM-114, Computer Science Department, Stanford University, (1970/4)