解 説

# XMLとコンピュータサイエンス

日本 IBM 東京基礎研究所
戸沢 晶彦 atozawa@jp.ibm.com
田渕 直 tabee@jp.ibm.com
立堀 道昭 mich@acm.org

XML の普及にともない，さまざまなデータを XML で統一的に記述し，これを処理したり交換したりすることが一般的になってきている．本稿ではこのような処理や交換に用いられる XML 変換の技術に注目する．XML 変換はまたコンピュータサイエンスのさまざまな分野と関連してくる面白いトピックである．その例としては，オートマトンや木変換器といった既存の理論に XML 変換に対する新しい応用が考えられたり，またいままでのプログラミング言語のものと異なる新しい型システムが必要とされたり，などということが挙げられる．本稿では特に形式言語やプログラミング言語の分野の知識を使って，XML の変換処理の技術とその最新のトピックをフォーマルに議論する．具体的には木変換器と正規表現型の話に触れる．そしてこれらによってどのように XML 変換の正しさを保証したり（XML 変換の型検査），また与えられた XML の変換処理をどのように最適化して効率のよいものにするのか（XML 変換の合成）ということを見てゆく．

## XML と XML 変換

計算機でデータを表現し，またやりとりするためにはそのデータの形式が決まっていなければならない．さまざまな計算機の間でデータをやりとりするためにはなるべく標準的な形式を決めたいという要求があったのであるが，そのための統一的なデータの形式として近年 XML が広く受け入れられてきている．

もっともこれは XML が何か斬新なデータ形式を持ち合わせているということではない，そもそも XML が表すものはいわゆる木構造データとみなせるものである．これはすなわち，構文木や，Lisp の S 式，あるいは入れ子データベースなど計算機で最もよくみられるデータ構造の 1 つである．これらの木構造を扱う技術はこれまでプログラム言語，データベースといった応用ごとに独自に発展してきた．しかし XML の普及に従いこれらの技術を XML を対象とした技術として統一的に扱うことができるようになってきた．

本稿ではこのような XML を扱う技術のうち特に XML 変換の技術を解説する．狭い意味の XML 変換とはある XML を入力として受け取り，別の XML を出力とするような計算のことである．このような XML 変換はそれ自体，**図 -1** のような主にデータ形式の間の変換に関する応用がある．この図で「検索」とは XML データベースの中を調べ，得られた結果を XML として持ってくるようないわゆるネイティブ XML データベースに対する検索のことを意味している．またスタイルシートは XML 形式のデータを Web ブラウザで閲覧できるような HTML 形式にフォーマットするような変換処理を行う．さらにより広い意味でとらえると XML 変換とは，計算機の別のデータ形式と，XML との間の変換を含む

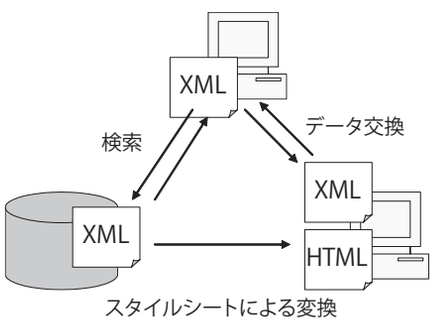


**図 -1 XML 変換の使われ方**

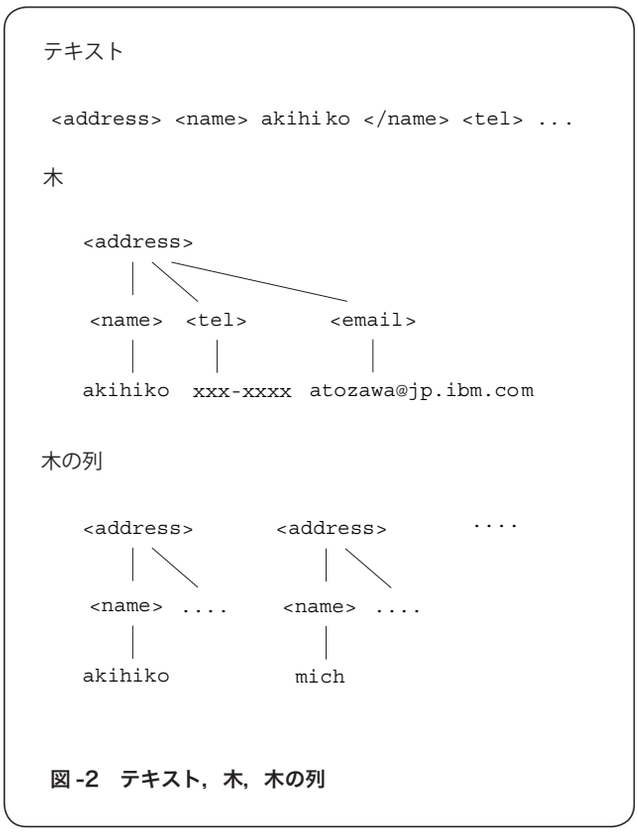


**図 -2　テキスト，木，木の列**

と考えることもできる．本稿でこれから見てゆく技法，特に正規表現型に基づいた型検査の技法はこのような広い意味の XML 変換に対してもほぼ同様に適用できる．

これから，XML のモデルおよびそれに基づいた XML 変換のモデルを与え，その後このモデルを使って XML 変換の型検査および合成を説明してゆく．1 つ注意しておく点として，この記事ではコンピュータサイエンス的な視点からみて分かりやすいモデルを考えているため，必ずしも XML あるいは具体的な XML 変換言語の仕様どおりのモデルを与えているわけではない．たとえば厳密に仕様に基づいた XML は必ず 1 つのルートノードを持つ木であり，また属性や名前空間など本稿では触れないものを含んでいる．

## ✦木と木の列

XML は他のさまざまな計算機の言語と同じように単語 (token) の並びからなるテキストで表現される．いま XML の中で使われる単語の種類は開きタグと呼ばれる $\langle a \rangle$，閉じタグと呼ばれる $\langle /a \rangle$（ただし $a$ は適当なタグ名）あるいはそれ以外の任意の文字列 $\boldsymbol{s}$ のどれかであるとしておこう．どんな XML もこの 3 種類の語の並びからなるテキストで表される．ただし，どんなテキストでも XML とみなせるわけではない．つまり XML が表すのは木構造として正しい形をしたテキストのみである．

このことを以下のような文法で表す．

$$\mathrm{XML} \ni v ::= \boldsymbol{s} \mid \langle a \rangle v_1 \cdots v_n \langle /a \rangle$$

この文法が表すことは (1) 文字列 $\boldsymbol{s}$ は XML である．(2) もし $v_1, \cdots v_n$ が XML ならばこの $v_1$ から $v_n$ までの列を $\langle a \rangle v_1 \ldots v_n \langle /a \rangle$ のように開きタグと閉じタグで囲んだものも XML である．ということである．これはいいかえれば XML が木構造を表現するということをいっている（**図 -2** 参照）．

しかし，XML についてもう 1 つの見方もある．それは XML は木の列であるとすることである．つまり上の $v \in \mathrm{XML}$ に加えて，$v_1 \ldots v_n$ という列もまた XML であるとすることである．この見方の方が都合がいいことがある．たとえばほとんどの XML 変換言語は 1 つの木だけでなく木の列を入力としたり出力としたりすることができる．

そこでこの記事ではこの「XML は木の列である」という見方をとる．このためにいま $v_1 \ldots v_n$ もまた XML である，というルールを追加してもいいのであるが，もう少し単純なルールにもできる．まず空の木の列を表す表現がいるのでこれを () で表す．

$$\mathrm{XML}' \ni v ::= () \mid \boldsymbol{s}\, v \mid \langle a \rangle v \langle /a \rangle v$$

この文法が表すことは，(1) 空の列は XML である．(2) 文字列 $\boldsymbol{s}$ から始まる列は XML である．(3) ノード $\langle a \rangle v \langle /a \rangle$ から始まる列は XML であるということである．

## ✦ XML 変換のモデル

では XML 変換のモデルを考えてみよう．一般に計算機の上で木構造データを処理するときに必要なのは再帰的な処理である．たとえば最も簡単な処理として木の複製ということを考えてみよう．複製なので，入力の木の列の各要素に対して新しい要素を作る処理を再帰的に繰り返すことになる．いまこのことを以下のような書き換えルール (1-3) によって表現しよう．

(1) $copy\ (\langle a \rangle\ v\ \langle /a \rangle\ r) \quad \rightarrow \quad \langle a \rangle\ copy\ (v)\ \langle /a \rangle\ copy\ (r)$
(2) $copy\ (\boldsymbol{s}\ r) \quad \rightarrow \quad \boldsymbol{s}\ copy\ (r)$
(3) $copy\ (()) \quad \rightarrow \quad ()$

**図 -3** にこれらのルールを使ってどのように木の列を複製するかを説明する．いま $copy$ に `<name>akihiko</name>` と `<tel>xxx-xxxx</tel>` からなる木の列を引数として与えるとする．このような木の列は $a$= `<name>`，$v$= `akihiko`，また $r$ = `<tel>xxx-xxxx</tel>` としてやれば，ルール (1) の → の左辺にマッチするので，まず

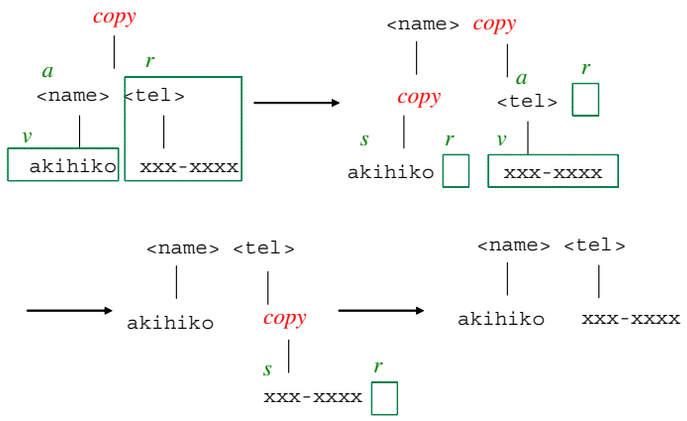


**図 -3　copy による XML 変換の動作**

このルールを用いる．すると (1) の右辺に各変数の値を代入したものつまり

<name> *copy* (akihiko) </name>
*copy* (<tel>xxx-xxxx</tel>)

というのが次のステップで得られる結果になる．同様に今度は 1 つめの *copy* はルール (2) を用い，2 つめの *copy* はまたルール (1) を用いて計算を進める．結果的に *copy* は元の木の列の複製を返すことが確かめられる．

さて以上では *copy* を例として XML 変換の動作を直観的に説明してきたが，では一般の場合はどうだろうか？特にこのような書き換えルールの → の左辺と右辺に何が許されるのだろうか．いま書き換えルールは $f(p) \to t$ という形をしている．この $f$ つまり *copy* などのことを関数記号と呼ぶ．一方 $p$ はパターンなどと呼ばれる．$p$ は木の列であって中に変数 $a$，$s$，$v$，$r$ などの出現[☆1] を許すものであるとすることができる．

では書き換えルールの右辺 $t$ には何が許されるだろうか．実はここに何を許すかによって，どんな変換が可能かが本質的に違ってくる．

- $t$ は木の列であって，ただし中に 1 引数の関数呼び出し $f(x)$（ただし $x$ は変数）の出現を許すもの．
- $t$ の中に任意のネストした関数呼び出しの出現を許すもの．たとえば $f(g(x), h(y))$ などのようなものも許す．

☆1 より詳しくいうと $a$ はタグ名，$s$ はデータの上を動く変数．また $v$ や $r$ は任意の木の列の上を動く変数とする．

前者のような書き換えルールだけを許すものを特に木変換器 (tree transducer) という．木変換器とはおおまかにいえば一度出力された木は二度と関数の入力として再利用されることはないような木の変換処理を行うものである．一方，後者のようなものを使えば原理的にいえば計算可能なものは何でも計算できることが知られている．つまり後者の形のものは 1 つのプログラム言語（関数型言語）を定義していると考えることができる．

以降で説明される型検査の手法は後者の形の XML 変換，つまり一般のプログラム言語と同じように強力な XML 変換にも適用できるものである．一方，合成の手法は一般には前者の制限された木変換器にしか適用できない．

## XML 変換の正しさの検査

### ✦ XML の型

XML の世界にはスキーマあるいは DTD と呼ばれる文書の型を定義する仕組みがある．

以下は XML の型の例である．

*Person* ::= <person>*Name Tel*? *Email*</person>
*Name* ::= <name>*String*</name>
*Tel* ::= <tel>*String*</tel>
*Email* ::= <email>*String*</email>

つまり XML の型とは木の具体的な列の代わりにその列の満たす正規表現，つまり和 ∪，繰り返し *，1 回以内の出現 ? などの演算子を用いて列の性質を書くことを許したものである．この型のことを正規表現型 (regular expression type) あるいは正規木文法 (regular

tree grammar) などとも呼ぶ（**図 -4** のコラム参照）．さらなる詳細については 文献 8）のチュートリアルなどが参考になる．

さて大文字の斜体 *Person*, *Name* などを型変数という．上の例のように木の上の正規表現の代わりにこれらの型変数の上の正規表現を書くこともできる．また型変数 *String* は任意の文字列 ***s*** の型を表すとしておく．

たとえば以下の文書は *Person** のインスタンスとなる．

```
<person>
 <name>akihiko</name>
 <tel>xxx-xxxx</tel>
 <email>atozawa@jp.ibm.com</email>
</person>
<person>
 <name>mich</name>
 <email>mich@acm.org</email>
</person>
```

: *Person**

正規表現型が表すのはある木の列の集合（言語）である．たとえば *Entry** に合致するような木の列は無数にあるから，この型の言語は無限集合になる．しかしあるクラスの言語ならばこのような無限集合を有限の方法で記述（計算機で表現できる記述）できる．このような言語のクラスのうちで，最も基本的でかつよい性質を持ったクラスが正規集合のクラスである．オートマトンとは正規集合を有限の方法で記述するためのあるデータ表現である．特にいま木の（列の）集合を考えているので「木オートマトン」というものを使う．正規集合のクラスの持つよい性質として，集合演算に対する閉包性や，空集合判定の決定可能性がある．実際，XML の型検査で使われる 正規表現型の部分型判定 $S <: T$ とは型 $S$ の言語が型 $T$ の言語に集合として包含されていること $[\![S]\!] \subseteq [\![T]\!]$ の判定である．このことは $[\![S]\!] \cap \overline{[\![T]\!]} = \emptyset$ とも書ける（$\cap$ は集合の共通部分，$\overline{\ }$ は補集合）．これは正規集合に対する集合演算や空集合判定を使えば決定できる．このような演算は木オートマトンを操作する手続きによって実現される．

**図 -4　正規表現型と木オートマトン**

## ✦型検査

型検査 (type check) の問題とはある XML の変換処理と，その変換の満たすべき型に関する制約を与えられたときに，その制約が本当に満たされるかを調べる問題である．つまり XML 変換処理が期待どおりの動作をするかを調べる 1 つの手法といえる．型検査やそれに基づいた型つきのプログラミングはたとえば C や Java によるプログラミングを考えれば分かるようにデバッグを助けるし，また型のインタフェースだけを公開するモジュールプログラミングや分散プログラミングで有効である．

いま次の変換を考える．

```
entry( <person>                  →  <entry>
         <name>n</name>               <name>n</name>
         <tel>t</tel>                 <tel>t</tel>
         <email>e</email>          </entry>
       </person>                   entry(r)
       r )

entry( <person>                  →  entry(r)
         <name>n</name>
         <email>e</email>
       </person>
       r )

entry(())                        →  ()
```

これは電話番号と電子メールアドレスを持ったアドレス帳から電話番号のみのリストを作成するような変換であるといえる．いま電話番号のみのエントリは以下のような型で書ける．

*Entry* ::= `<entry>`*Name Tel*`</entry>`

では上の変換が実際にアドレス帳 *Person** から電話番号リスト *Entry** を作るような変換であるかを調べてみよう．つまり問題は

*entry* : *Person** → *Entry**

のような問題である．このことは以下のアルゴリズムで調べられる．ただし**アルゴリズム 1** の中では型の包含関係 $S <: T$ を調べるアルゴリズムおよび型推論のアルゴリズムを使っている．これらのアルゴリズムはかなり高度なので詳細は本稿では省略する．しかしそのベースとなる木オートマトンなどの理論的背景については図 -4 のコラムを参照のこと．

**アルゴリズム 1**　型検査が成功することは要するに *entry* を定義する各ルールが *Person** →*Entry** という制約を満たすことに帰着される．いま () → () のルールなどは簡単である．つまり出力として得られるのは常に () であ

るが，これは *Entry** のインスタンスだからである．いまこのことを () <: *Entry** という包含関係が成り立つといってもよい．この $S <: T$ とは $S$ のインスタンスは必ず $T$ のインスタンスであるという意味である．

一方，左辺に変数が含まれるルールたとえば

$$entry\left(\begin{array}{l}\texttt{<person>}\\ \quad\texttt{<name>}n\texttt{</name>}\\ \quad\texttt{<tel>}t\texttt{</tel>}\\ \quad\texttt{<email>}e\texttt{</email>}\\ \texttt{</person>}\\ r\end{array}\right) \rightarrow \begin{array}{l}\texttt{<entry>}\\ \quad\texttt{<name>}n\texttt{</name>}\\ \quad\texttt{<tel>}t\texttt{</tel>}\\ \texttt{</entry>}\\ entry(r)\end{array}$$

などについては，変数の型をまず推論しなくてはならない．推論の結果は

$n, t, e : String$
$r : Person*$

などとなる．いま *entry* は *Person** → *Entry** という関数だと仮定しているので，$entry(r) : Entry*$ である．これと $n$, $t$ の型から，

$$\left(\begin{array}{l}\texttt{<entry>}\\ \quad\texttt{<name>}String\texttt{</name>}\\ \quad\texttt{<tel>}String\texttt{</tel>}\\ \texttt{</entry>}\\ Entry*\end{array}\right) <: Entry*$$

を調べることになるが，これは成立する．

さらに型検査には変換結果の正しさの検査のほかに，網羅チェック (exhastiveness check) と余剰チェック (redundancy check) というものがあり，前者は入力木が必ずどれかのパターンにマッチすること，後者はパターンに重複がないことを調べる．このようなチェックは型の包含などを用いて容易に可能であるが，この記事では詳細には触れない．

## XML変換の合成と最適化

いまXMLは数Mbyteやそれ以上の大規模なデータをも扱うようになってきている．このようなときにXML変換を効率的に行うためにはなるべくメモリを使わないことが，より具体的には処理の途中で「中間木を作らない」ということが大切になってくる．このようなときに使える手法が木変換器の合成 (composition) あるいは関数の融合 (fusion) と呼ばれる手法である．

いまここでは2つの木変換器 $f$, $g$ が与えられたときにその合成を計算する方法を説明する．ある木 $v$ に対して $f$ と $g$ の2つの変換処理を木変換器を順次動作させることで行う場合，まず $f(v)$ の結果をメモリに保持しておき，この結果に対して次に $g(f(v))$ を計算する必要が生じる．そこで $g(f(v))$ を2つの木変換によって計算するのではなく1つの木変換 $(g \circ f)(v)$ で計算できるならば $f(v)$ の結果をメモリに持つことなく計算ができるのではないかというのがアイディアである．このように2つの木変換器 $f$, $g$ から木変換器 $g \circ f$ をさきに作っておくことを木変換器の合成という．

いま木変換器でただし，書き換え規則 $f(p) \rightarrow t$ の，右辺 $t$ の中の関数呼び出しが最後尾 (tail position) にのみに現れるものを考えよう．つまり関数呼び出しが $t$ を定義するある木の列の中に現れるときには常にその右端の要素として現れるということである．たとえば *entry* を定義する各ルールでは，$entry(r)$ は最後尾に現れている．このような木変換器については以下のような合成アルゴリズムが知られている．

**アルゴリズム2**　いま変換 $f$ と変換 $g$ の合成 $g \circ f$ を考えよう．変換 $f$ を定義する各書き換えルール $f(p) \rightarrow t$ に対して次のように両辺に $g$ を適用したものを考えてみる．

$$(g \circ f)(p) \rightarrow g(t)$$

となる．ここでこの $g(t)$ を前もって計算しておくことができるならば新しい木変換 $g \circ f$ のルールが定義できたことになる．たとえばさきほどの変換 *entry* と**図-5** の *search* との合成を考えてみる[☆2]．

いま *entry* の次のルールを考える．

$$entry\left(\begin{array}{l}\texttt{<person>}\\ \quad\texttt{<name>}n\texttt{</name>}\\ \quad\texttt{<tel>}t\texttt{</tel>}\\ \quad\texttt{<email>}e\texttt{</email>}\\ \texttt{</person>}\\ r\end{array}\right) \rightarrow \begin{array}{l}\texttt{<entry>}\\ \quad\texttt{<name>}n\texttt{</name>}\\ \quad\texttt{<tel>}t\texttt{</tel>}\\ \texttt{</entry>}\\ entry(r)\end{array}$$

というルールについてこの右辺に *search* を適用したものを計算してやると，これは *search* の最初の規則にマ

☆2 *search* には *if* 式がでてくるが，このような *if* 式があっても合成は可能である．これは $f(\textit{if } e \textit{ then } s \texttt{ else } t) \equiv \textit{if } e \textit{ then } f(s) \texttt{ else } f(t)$ という等価関係が成り立つからである．

$$search\left(\begin{array}{l}\texttt{<entry>}\\ \texttt{<name>}n\texttt{</name>}\\ \texttt{<tel>}t\texttt{</tel>}\\ \texttt{</entry>}\\ r\end{array}\right) \rightarrow \textit{if } n{=}\texttt{mich } \textit{then } t \textit{ else } search(r)$$

$$search(()) \rightarrow \texttt{None}$$

**図 -5　*Entry** の上で `mich` の電話番号をさがすような処理**

$$(search \circ entry)\left(\begin{array}{l}\texttt{<person>}\\ \texttt{<name>}n\texttt{</name>}\\ \texttt{<tel>}t\texttt{</tel>}\\ \texttt{<email>}e\texttt{</email>}\\ \texttt{</person>}\\ r\end{array}\right) \rightarrow \textit{if } n{=}\texttt{mich } \textit{then } t \textit{ else } (search \circ entry)(r)$$

$$(search \circ entry)\left(\begin{array}{l}\texttt{<person>}\\ \texttt{<name>}n\texttt{</name>}\\ \texttt{<email>}e\texttt{</email>}\\ \texttt{</person>}\\ r\end{array}\right) \rightarrow (search \circ entry)(r)$$

$$(search \circ entry)(()) \rightarrow \texttt{None}$$

**図 -6　合成結果**

ッチするので[☆3]

$$\textit{if } n{=}\texttt{mich } \textit{then } t \textit{ else } search(entry(r))$$

というように変換される．ここでこの $search(entry(r))$ は $(search \circ entry)(r)$ に置き換えてよいので，新しく作った合成関数による表現に直せた．

このアルゴリズムが完了するとたとえば $search \circ entry$ の結果は**図 -6** のようになる．

[☆3] もしどれにもマッチしないならば，右辺の $entry(r)$ をもう一度展開してやる．このとき $r$ がマッチするパターンに応じて場合わけをしてやることになる．

## 実世界における応用

いままで XML の型検査と XML 変換の合成による最適化という 2 つのトピックを説明してきた．ではこれらは実世界の XML 技術に対してどのような応用があるのだろうか？　実世界での XML 技術というものは多岐に渡る．まずさまざまなボキャブラリが XML の上に定義されている．そしてこれらのボキャブラリは型検査のところで説明したような XML の型によってその構文が詳細に規定されている．さらにこのように定義された XML データをどうやって変換または処理するかという問題がある．ここではこれらのさまざまな XML データ処理技術あるいはそのための言語について見てみる．

まず言語という点から見ると最近では XML 専用の言語あるいは既存言語の XML 拡張などが数多く提案され

てきている．これらの言語ではXMLのための組み込みのデータ型が用意され，またXMLの変換あるいは処理のための組み込みの演算（つまりXMLの構築，検索やパターンマッチなどの演算とそのための構文）が用意される．このような一級のオブジェクト(first-class object)としてXMLを扱う言語では，XMLの型に関して今回説明したような型検査を行うことが普通である．このような検査によってプログラムがエラーを起こさないことや，またその出力が常に正しい型のXMLであることが保証されるわけである．今回説明した正規表現型に対する型検査のやり方はXML変換言語XDuce[6]で使われたものを簡略化して説明したものである．XDuceの影響を受けた型検査を備えた言語はJWIG[3]，Xtatic[5]，Cω[2]，XJ[4]など多くのものがあり，また現実に使われている言語でもXQueryは型検査を備えている．これらの言語はXML処理の組み込みの演算にそれぞれ違いがあるが[☆4]，扱う型は今回説明したものとほぼ同じである．もう1つCDuce[1]という言語があるが，ここでは正規表現型と同じ考え方による部分型判定（意味論的な部分型判定 = semantic subtyping）を関数型にまで拡張している．このようにプログラム言語の研究分野では正規表現型あるいは意味論的な型というのが面白いトピックになっている．

またXMLの1つの重要な応用としてXMLデータベースとそれに対する検索がある．この検索のためにはXPathというパス式，さらにこれを元にしてさまざまな拡張を加えたXQueryという検索言語を使うというのが主流である．さてデータベースの世界ではもとから検索式の合成あるいは検索式の展開(query unfolding)という最適化が重視されてきた．このような技術は自然にXQueryなどのXML検索言語に対しても拡張され応用されてきている．その内容は基本的には説明したような木変換器の合成手法と同じものである．その効果はたとえばビュー検索式の合成のようなときに，中間のビューを作らずに直接ほしいビューを作るなどの処理の最適化である．

さらにWebサービスのプログラミングなどのXML変換に目を転じてみると，XMLは通信のメッセージとして使われている．このような場合のXML変換を高速に行うために特に必要なのは「ストリーム処理」にもとづく最適化である．いままで本稿ではXMLは木構造であるとして議論してきたが，XMLをテキストつまり文字のストリームとして処理するのがストリーム処理である．今回の話では木変換器の合成を扱ったが，ストリームから木を作るパース処理をある変換器とみたてて，これと実際の木の変換を合成することにより，まったく木をメモリ上に作らずにテキストをストリーム処理するだけで変換を実現するという研究もある．XTiSP[7]という実験的な言語などがその結果である．**図-7**にこのようなストリーム最適化のイメージを示す．この手法を実現するためには今まで述べたような木変換器だけでは不足でスタックつき木変換器という特殊なものが必要になる．また関数型言語の分野でも融合と呼ばれる同様の手法を用いてXML変換の最適化をする研究が行われてきている．ストリーム処理では通信で得られたXMLストリームをそのまま逐次処理することによって結果が得られるため，XML木をメモリ上に作らずに処理をすることが可能であり，また出力についても処理が済んだところから出力してゆくことが可能である．このため，この最適化に基づきWebサービスのレスポンスやスループットなどを向上することができるのではないかと期待されている．

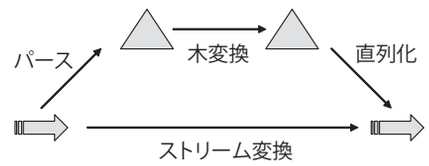


**図-7　XTiSP: 合成によるストリーム最適化**

☆4 JWIG，Xtatic，Cω，XJでは今回説明したような変換だけではなく，データの更新(in-place update)の操作が許されているが，たとえこのような操作があっても型検査のアルゴリズムはほぼ同じである．

**参考文献**

1) CDuce, http://www.cduce.org/
2) Comega, http://research.microsoft.com/Comega/
3) JWIG JavaTM Extensions for High-Level Web Service Development, http://www.brics.dk/JWIG/
4) The XJ Project, http://www.research.ibm.com/xj/
5) The Xtatic Project, Native XML processing for C#, http://www.cis.upenn.edu/~bcpierce/xtatic/
6) XDuce, A Typed XML Processing Language, http://xduce.sourceforge.net/
7) XTiSP, An Implementation Framework for XML Transformation Languages Intended for Stream Processing, http://xtisp.psdlab.org/
8) 村田　真，川口耕介：XMLとスキーマ言語，コンピュータソフトウェア，21(6)，pp.472-481（Nov. 2004）．

（平成17年8月15日受付）