XMLストリーム処理に対する型検査およびノード補間の技法

戸沢 晶彦*
日本 IBM 東京基礎研究所

萩谷 昌己†
東京大学情報理工学系研究科

要旨

IBM alphaWorks で公開されている型つき XML 処理言語 XML Processing Plus Plus (XP++) の言語仕様の特徴とその処理系についてのべる. 既存の型つき XML 処理言語である XDuce, JWIG, Xtatic などと比べ, XP++ はストリーム入出力に基づいた XML 処理言語であるという点で異なっている. さらに本稿はこの XP++ の処理系のうち, ノード補間機能とよぶ部分をどう実現したかに焦点をあて, この部分を詳しく説明する.

1 はじめに

XML (eXtensible Markup Language) はタグづけされることによって木構造を持つようになったテキスト文書である. XML の普及に伴い, XML のアプリケーションを記述するのに適したプログラミングスタイルはどんなものであるかという議論が盛んである.

XML といままでのプログラミング言語で取り扱ってきたデータと大きく違うのはそのデータモデルである. XML のデータモデルとしては DTD (Document Type Definition = 文書型定義), そしてその拡張であるスキーマ (XML Schema, RELAX NG) などが使われる. これらのデータモデルは抽象データ型やオブジェクト型のようないままでのプログラミング言語でよく使われてきたいわゆる「型」のようにみなせる部分もあるが, 大きくことなる部分もある.

細谷らは XML が表現する木構造をそのまま, 値として取り扱う型つきプログラミング言語 XDuce を考案した [5, 6]. 彼らは型として DTD やスキーマと同等の表現力がある正規表現型を新しく考案しこれをもちいた. このような型つき言語が XML のプログラミングに対して与える便宜はいままでの型つきプログラミング言語のものと同じである. すなわち型検査の利用によりプログラムの動作のある面での正しさが保証でき, 型エラーの報告によりデバッグがしやすいということである.

*atozawa@trl.ibm.com
†hagiya@is.s.u-tokyo.ac.jp

著者のひとりは最近 XDuce とおなじく XML の文書型 (すなわち DTD, スキーマ = 正規表現型) に基づいた型検査を行う新しい言語 XML Processing Plus Plus (XP++) を開発し, これを公開した (`http://www.alphaworks.ibm.com` にて公開). 本稿はこの XML 処理言語 XP++ の言語仕様の特徴とその処理系について説明するものである.

XP++ がいままでの型つき XML 処理言語と異なるのはそれがストリーム入出力の概念に基づいて設計されたことである. つまりいままでの型つき XML 処理言語は XML が表現する木構造をそのままプログラムが操作するようなデータ構造とみなしていた. しかし, もう一方の見方をすれば, XML はテキストであり, また HTTP などを介して通信されるメッセージでもある. テキストやメッセージの処理にはストリーム入出力 (標準入出力, ファイル入出力) が使われてきた. ここでいうストリームとは前から後ろへ順に読み込む, あるいは書き出すという使いかたをされるものである. このためストリームに基づいた XML の処理ではそう複雑なことはできない. しかし一方でストリームは取り扱いがたやすく, また高速である. このことはそのまま XP++ の利点である.

さらに, われわれは XP++ に独自の機能として, 出力ストリームに対するコンパイル時のノード補間という機能を付け加えた. この機能は, ストリームに対して宣言された文書型に応じて, プログラムでは指定されていないタグ構造をストリームに出力するような機能である. このような機能を導入した背景にはプログラムの記述をさらに簡潔にしたい, 文書型の改変に強いプログラミングをしたい, あるいは文書型を型としてだけではなく「テンプレート」として用いたいなどという理由があった. 一方, 技術的にはこれは XDuce などの既存技術では解決できない難問である. 本稿の後半ではこのノード補間問題を XP++ の処理系はどのように解いているかに焦点を当てて説明する.

関連研究

XDuce [5, 6] の型システムは, XQuery Formal Semantics [11] や Xtatic [10] にも影響を与えた. XDuce と

XQuery は関数型の言語である. ただし XQuery は SQL の影響を強く受けている. XDuce は ML に似たパターンマッチを用いて XML を読みこむのに対し, XQuery は XPath を用いる. XQuery Formal Semantics は XQuery のための型検査の枠組である. ここで XPath とは W3C で定められた仕様であり, XML の検索のためにパス式を用いるものであり, 広く使われている.

Xtatic は C# に対して XDuce と同様のパターンマッチによる XML 処理の機能をつけた言語である. XDuce と同様の型検査を行う.

JWIG [3] もまた文書型に基づいた型検査を備えた言語であり, gap という穴を持った文書の断片をつなぎあわせることにより, XML 文書を作成する機能を備える. JWIG はいまのところ XML 入力の機能はないが, XPath を用いてこれを行い, 入出力をあわせて型検査を行うことも考えているようである.

X-Prolog [4] はやはり型検査を行う, 論理型の XML 処理言語である.

ストリームに基づいた XML 処理系としては, まず XML のストリームに基づいたパーサとして SAX パーサが非常によく使われている. また, XPath をストリームに基づいて処理するという研究 [2, 1] や実装は多数ある. 一方, 中野ら [8] は XML の変換器の仕様から属性文法の合成の手法を用いて, ストリームに基づいた変換器を自動生成するという研究を行った.

本稿の構成

本稿の構成は以下のようである. まず次節で XP++ でのプログラミングを説明し, 他の XML 処理言語とのよりくわしい比較をする. 3 節では, ノード補間問題に焦点を当て, この問題を定式化し, さらに 4 節でその解法の提案をくわしく説明する. 5 節はまとめである.

2 XP++ の言語仕様

2.1 XP++のプログラム例

図 1 に XP++ でのプログラミングの例をしめす. このプログラムは次のような XML 文書が標準入力に与えられたときに, これを XHTML のテーブル形式に変換するものである.

```
<address>
   <person>
      <name>Akihiko Tozawa</name>
      <email>atozawa@trl.ibm.com</email>
   </person>
   <person>
      <name> ..... </name>
      <email> ..... </email>
   </person>
   ....
</address>
```

```
XmlOut html = new Html(System.out).html;

XmlOut head = html.head;
head.title = "ADDRESS BOOK";

XmlOut body = html.body;
XmlOut table = body.table;
table.border = 1;
XmlOut tbody = table.tbody;

XmlIn address =
     new Address(System.in).address;

while (true) try {
    XmlIn person = address.person;
    XmlOut tr = tbody.tr;
    tr.td = person.name;
    tr.td = person.email;
}
catch (XmlIn.EmptyException e) { break; }
```

図 1: XP++プログラムの例

プログラムの動作を説明する. まず 1 行目および 11 行目にある new は新しい XML ストリームを作る命令である. ここで Html や Address はクラス名ではなく文書型の定義を示している. 文書型定義は例えば次のような DTD ファイル Address.dtd である.

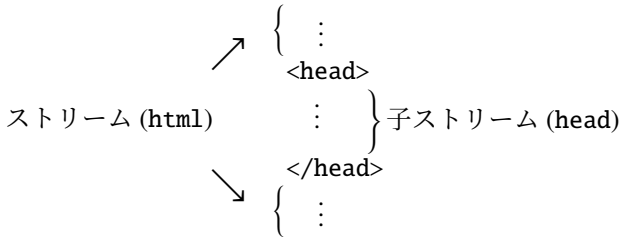
```
<!ELEMENT address (person)*>
<!ELEMENT person (name, email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

このような文書型定義はプログラムのコンパイル時に必要となり, 作成された XML ストリームが従うべき, XML 文書の形式をしめしている. 冒頭の XML 文書はこの Address.dtd の形式にのっとった文書である.

XP++ における入出力命令は基本的には

ストリーム変数 = ストリーム.ノード名;

という形をしている. たとえば XmlOut head = html.head; では, 出力ストリーム html に head というノード (= XML 用語でいう要素 (element) あるいは属性 (attribute)) を書き出す. そして, ストリーム変数 head には head タグに囲まれた部分に相当する子ストリームが代入される. 図示すると以下のようである.



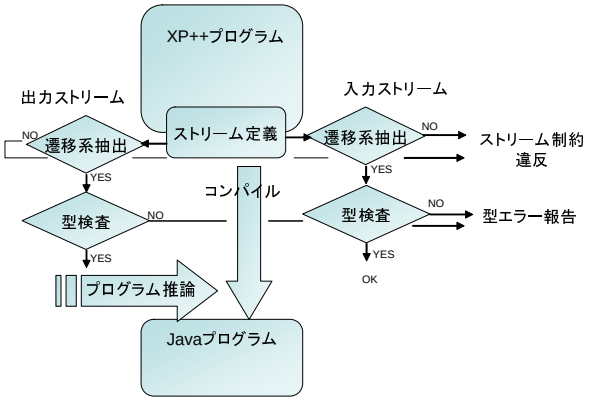


図 2: XP++ の処理系

入力命令も同様である. 例えば `XmlIn person = address.person;` は入力ストリーム address からひとつめの person ノードの中身を取り出す命令である. また, 入出力命令を組み合わせた `tr.td = person.name;` は name ノードの中身であるテキスト値 (Java の String オブジェクト) の内容を td ノードの中に書き込むという動作を意味する.

入力命令では読み込もうとするノードが見付からなかったときには, 例外 (XmlIn.EmptyException) が起こる. 13 - 19 行目の try 節はこれを処理するためである. 結果としてこのプログラムは例外がおこるまで, 順次, 全ての person 要素を読み込んでいくという動作をする.

2.2 XP++の特徴

XP++ には (1) 文書型に基づいた型検査, (2) ストリーム API, (3) Java の拡張, (4) シンタクスの簡潔さ, のような特徴がある.

文書型に基づいた型検査

XP++ は XDuce と同様に正規表現型を用いた型検査を行う. たとえば図 1 のプログラムは, Address 型の文書を読み込み, これを Html 型の文書に変換するものである. このプログラムが型検査を通れば, これが出力する文書は全て Html 型に合致したものとなる. また, 入力ストリームの型に関しては捕獲されない XmlIn.EmptyException はないという性質が検査される. さらに, もし検査で「常に XmlIn.EmptyException を投げる」ような命令が見つかった場合, これも型エラーとしている.

ストリーム API

XDuce, JWIG, Xtatic などは XML 文書を木構造としてみる視点にたった言語であるといえる. XDuce などでは XML をプログラムがとりあつかうような汎用のデータ構造とみなし, たとえばこのデータ構造上でソートその他の処理を記述することもできる.

Java では XML 文書を全てメモリ上に木構造として展開するような API として DOM がよく使われる.

これらとは違い XML をたんなるタグ情報の付加された文字列ストリームであるとして処理するのがストリーム API である. ストリーム API は DOM などに比べ, 原始的であるといえるが, そのかわり,

- シンプルである
- 高速, 高メモリ効率である.

という決定的な長所がある. ストリーム API のシンプルさは, 代表的なストリーム API である SAX や本研究が証明している. ストリーム API が高速, 高メモリ効率なのはもちろん, 中間的なデータ構造をメモリ上に作らないで処理するからである. ストリーム API を使って XML を出力するという手法はいままであまり実用では使われなかったが, XP++ では XML の入力はもちろん XML の出力もストリーム (たとえば標準出力ストリームに対する print 命令) による.

Java の拡張

XP++ は Java 言語の拡張として実装されている. XP++ 言語では XML ストリーム型として宣言された値の取り扱いのみが新しくつけ加わった部分である. XP++ の処理系は, ソースプログラムを通常の Java プログラムにコンパイルする. XML の出力に関しては Java の出力ストリームをそのまま使うことができる. XML 入力のためには XML のパーサのライブラリが必要である.

XP++ が提供する XML 操作命令は命令型言語に適したものであるといえる. たとえば XDuce はパターンマッチ機構を使って XML 文書の内容を取り出すような API を使っている. XDuce のパターンマッチは関数型言語 ML のものと非常に似ており, 関数型言語である XDuce としては自然である. いっぽう最近提案された命令型言語 Xtatic ではこの XDuce のものと同等のパターンマッチを命令型言語の switch 文として実現しようとしている. しかし, Xtatic の構文は命令型言語としては複雑なように感じられる.

JWIG も XP++ と同様 Java の拡張である.

シンタクスの簡潔さ

現在よく使われている Java 上の XML 処理の API としては DOM, SAX, などがある. DOM はノードの取り出し, 生成, 挿入, 削除などに関する多数のメソッド定義を持った数十のクラス定義からなる巨大な API である. SAX は DOM よりは遥かにシンプルであるが, それでも多数のメソッド定義からなる.

いっぽう XP++ で必要となる命令は基本的にはストリーム変数 = ストリーム.ノード名; という形のも

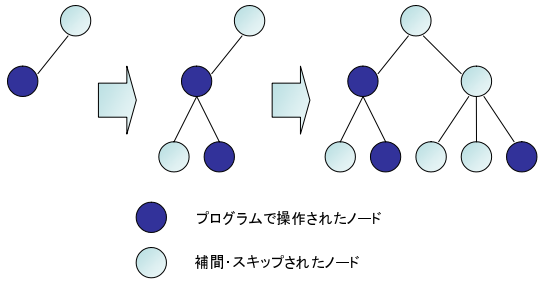


図 3: ノードのスキップおよび補間

のひとつだけである. これは同様に命令型言語の拡張である Xtatic や JWIG と比較しても非常に簡潔である.

2.3 ストリーム制約

XP++ では以下のようなプログラムは認めない.

```
(1)   XmlOut div = body.div;
      body.p = "p in body";
      div.h1 = "h1 in div";

(2)   XmlIn div = body.div;
      while (...) {
         div = div.div;
         ....
      }
```

図 2 に XP++ の処理系の仕組みを示した. 上のようなプログラムはこの図の処理の流れで「ストリーム制約違反」と判定されるものに相当する.

(1) のようなプログラムが違反である理由はこのようなプログラムはストリームの処理の順序で実行することができないからである. 一方, (2) のようなプログラムはループをまわるたびに div のネストが深くなっていく, このようなものも違反とする. このストリーム制約についての詳細は 3.3 節で述べる.

2.4 XML 出力に対するコンパイル時ノード補間

最後にノードのスキップおよび補間を説明する. じつは XP++ の入力ストリームの操作ではノードのスキップという処理が行われ, もう一方の出力ストリーム操作ではノードの補間という処理が暗黙に行われる.

まずこれがどのような処理なのかを説明する (図 3 参照). たとえば <html><head><title>ADDRESS BOOK</title></head>...</html> のような文書が作りたいとする. このとき実は図 1 のように明示的にすべてのノード生成を書く必要はない.

次のように書いても図 1 のプログラムと同等の文書を作ることができる.

```
XmlOut html = new Html(System.out);
html.title = "ADDRESS BOOK";
...
```

上のように書くと XP++ のコンパイラが自動的に省略されたノード (html, head) に対するノード生成命令を挿入したプログラムを作ってくれる. これがノードの補間処理である.

同様に入力ストリームで address.name と書いたときにはこれは address ノードの子孫のどこかにある name ノードのことを意味する. これがノードのスキップ処理である.

ここではとくにノードの補間処理に焦点をあてる. もちろん XP++ ではこのような補間処理なしで図 1 のようにプログラムを記述することもできる. それなのにとくにノードの補間処理を導入した理由はまずプログラムを簡潔に書けること, そしてその他に以下の 2 つの点がある.

まず文書型を XML 文書のテンプレートとして使えないかということである. そもそも文書型とは XML 文書の仕様を示すものであり, (プログラム言語の型とは違い) たとえば省略された値のデフォルト値の定義なども文書型に含まれる. じっさい XP++ のノード補間処理によってデフォルト値を含んだノードを自動的に文書に挿入することができる.

次の点として, 文書型の更新に強いプログラムが書けるのではないかということがある. もちろん文書型が常にバージョンの互換性を保つように更新されていれば, もとより問題は起きないが, XP++ ではたとえば中間ノード (例の head ノードなど) を挿入するような更新に対しても, プログラムを変更する必要が生じない. とくに文書型の仕様自体の開発と並行してプログラミングが行われる場合には, プログラムをきっちり書かなくてもコンパイルを通るという補間機能は有用ではないかと予想される.

技術的にはノード補間処理はむずかしい問題である. じっさい XP++ の処理系のこの処理以外の部分は XDuce の技術あるいは命令型言語の解析などの既存技術を応用すれば難しくない. 3 節以降ではこのノード補間処理をどう実現するかに焦点をしぼり, 説明をしてゆく.

3 ノード補間問題の定式化

3.1 概略

ここでは XP++ のソースプログラム中の出力ストリームに関する部分を仕様プログラムとよび, コンパイル後のノード生成命令が挿入されたプログラムを補間プログラムとよぶ. ノード補間問題とは仕様プログラム, 文書型が与えられたときに, 常に文書型に合致する結果を出力するような補間プログラムのひとつを求めることができるかという問題である.

本節では文書型, 仕様プログラム, 補間プログラムおよびノード補間問題に対して形式的なモデルをあたえる. このモデルは XP++処理系の実装のもとになったものである. ただし, 次のような点における実際の処理系の仕組みはモデルに反映されていない.

- 文字列データの取り扱い
- 再帰呼び出しの考慮
- 属性の考慮

また, 実際は仕様を満たす補間プログラムの可能性はほとんど常に複数あるいは無数にある. そこでわれわれは

- なるべく小さな補間ですむような補間プログラムの推論戦略

を考える必要があるが, これに関しても今回は詳しくは述べられない. 最終的にはこれらの仕組みを全てふくむような定式化が望ましいが, すこし繁雑であり, またこれらの部分ではこれからまだ言語仕様の改変などもありうる. そこで本稿ではもっとも基本的であり, なおかつ XDuce などの既存技術との差分になっていると著者が考えるモデルを取り扱う.

とくに再帰呼び出しの定式化を省いていることが原因で, 本稿のモデル上ではプログラムは任意の深さの XML 文書を作ることができない. しかし実際には再帰呼び出しを含むような拡張はノード補間の問題自体と比べれば容易な問題である [1]. XP++ では再帰呼び出しを利用することにより深さの定まらない XML 文書を作るようなプログラムを書くことができる.

また, XP++ の実際の処理系では「文字列ノードの補間はしない」という制約がある [2]. 今回のモデルではこの制約は全く考慮されていない.

3.2 文書型のモデル

まず, XP++ で用いる文書型 (DTD, スキーマ, 正規表現型) のモデルを定義する. ここでは簡単のために, DTD や正規表現型などの定義ではなく, それを翻訳した内部表現を示す.

XML の文書型については細谷 [5], 村田ら [7, 9] など多くの研究や提案があり,「正規木言語」とよばれる言語クラスを用いるということが一般的になっている. 以下の内部表現 (2 分木文法) もこの「正規木言語」を定義するものである. 前節で触れた DTD の定義から以下のような 2 分木文法を取り出す手法はすでによく知られているため [5], ここでは略す.

2 分木文法は $(\mathcal{G}, \rightarrow, \mathrm{G}^{in})$ という 3 つ組みである.

- 記号 $\mathcal{G}$ は非終端記号の集合
- 開始記号 G^{in} $(\in \mathcal{G})$
- 生成規則 $\mathrm{G}_0 \rightarrow \langle\sigma\rangle\mathrm{G}_1\langle/\sigma\rangle\mathrm{G}_2$ とし G_1 および G_2 は終端記号 η でも非終端記号でもよい. また σ は任意の XML のタグ名である.
- 終端記号 η

η は空文字をあらわす唯一の終端記号であるとする. 本稿では説明を簡単にするために XML 文書中の文字列データはすべて空文字であるとしておく.

例 1. 以下のような文法を考える.

$$\begin{aligned}
&\mathrm{G}^{in} \rightarrow \langle\mathtt{r}\rangle\mathrm{G}^{\mathtt{x}}\langle/\mathtt{r}\rangle\eta,\\
&\mathrm{G}^{\mathtt{x}} \rightarrow \langle\mathtt{x}\rangle\eta\langle/\mathtt{x}\rangle\mathrm{G}^{\mathtt{ab}},\\
&\mathrm{G}^{\mathtt{x}} \rightarrow \langle\mathtt{x}\rangle\mathrm{G}^{\mathtt{x}}\langle/\mathtt{x}\rangle\mathrm{G}^{\mathtt{ab}},\\
&\mathrm{G}^{\mathtt{ab}} \rightarrow \langle\mathtt{a}\rangle\eta\langle/\mathtt{a}\rangle\mathrm{G}^{\mathtt{ab}},\\
&\mathrm{G}^{\mathtt{ab}} \rightarrow \langle\mathtt{b}\rangle\eta\langle/\mathtt{b}\rangle\eta,
\end{aligned}$$

この文法はたとえば次のような XML 文書を生成する. ただし <a/> は <a> の略とする.

```
<r>
    <x>
        <x/>
        <a/>
        <a/>
        <b/>
    </x>
    <b/>
</r>
```

以下では, ある非終端記号 G から生成される言語を $\mathcal{L}(\mathrm{G})$ で示す.

以上の定義は 2 分木オートマトンを定義しているとみなすこともできる. ただし, 本稿では XML 文書との関連性を明確にするために上のような生成文法による定義とした.

3.3 仕様プログラム

XP++ プログラムからある出力ストリームに対する操作をぬきだしたものが, 仕様プログラムである.

XP++ プログラムはループや条件分岐, 例外処理などの制御文を含んでいる. プログラムが与えられたとき, われわれはまずこの制御フローの構造を遷移系 $(S, \rightsquigarrow, S^{in}, s^{fin})$ として抽出する.

- 状態集合 S
- 初期状態 $S^{in} \subseteq S$
- 終了状態 $s^{fin} \in S$

[1] 再帰呼び出しを考えたとき, (1) 数えあげアルゴリズム (図 4 参照) (2) 推論アルゴリズムともに若干の拡張がいる. しかし推論アルゴリズムは数えあげの結果をうまく利用すれば (本稿のアルゴリズムと同様に) バックトラックなしで動作可能.

[2] 例えば XHTML には必ず `title` という要素が必要であり, この `title` ノードの中身は文字列 (2 節の DTD の例にある `#PCDATA`) でなくてはならない. よって `title` ノードへの文字列の代入を含まない XP++ プログラムは常に型エラーとなる.

- 遷移 $\rightsquigarrow \subseteq (S \setminus \{s^{fin}\}) \times S$

次にわれわれは出力ストリームに対するノード出力命令の列に着目する. ここでは各命令をアクションとよぶ. 以下に, 遷移系の各状態 $s\,(\in S)$ に対するアクションを定義し, これをアクション仕様 (α, d) と呼ぶ.

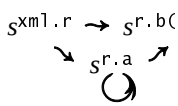
- α は, $S \setminus \{s^{fin}\}$ から <σ> の形の文字への写像
- 深さ $d \in S \to$ 自然数

写像 α は各状態において生成されなければならないノード名を示す. 深さ d はおおまかに言えばアクションがどのストリーム変数に対して実行されるかを示す値である. これについては以下の例を用いて説明する.

例 2. 以下のプログラムを考える.

```
XmlOut xml = new Example(System.out);
XmlOut r = xml.r;
while (...) r.a = "";
r.b = "";
r.b = "";
```

このプログラムから次のような遷移系を抽出することができる. 状態集合 $S = \{s^{\mathtt{xml.r}}, s^{\mathtt{r.a}}, s^{\mathtt{r.b(1)}}, s^{\mathtt{r.b(2)}}, s^{fin}\}$ とし, $S^{in} = \{s^{\mathtt{xml.r}}\}$ とする.



$$d = \{s^{\mathtt{xml.r}} \mapsto 1,\ s^{\mathtt{r.a}} \mapsto 2,\ s^{\mathtt{r.b(1)}} \mapsto 2,\ s^{\mathtt{r.b(2)}} \mapsto 2,\ s^{fin} \mapsto 0\},$$

$$\alpha = \{s^{\mathtt{xml.r}} \mapsto \text{<r>},\ s^{\mathtt{r.a}} \mapsto \text{<a>},\ s^{\mathtt{r.b(1)}} \mapsto \text{<b>},\ s^{\mathtt{r.b(2)}} \mapsto \text{<b>}\}$$

たとえば上のプログラムの場合, `r.a = "";` を実行するときに `r` および `xml` にバインドされたふたつのストリームが生きている. 変数 `xml` に代入されているのは `new` 命令で作られたストリームであり, この深さは 1 であるとする. 変数 `r` は `xml` から派生した子ストリームが代入され, この深さは 2 である. ここで, a-ノードを作るという $s^{\mathtt{r.a}}$ におけるアクションは変数 `r` に対して実行されるので $d(s^{\mathtt{r.a}}) = 2$ である.

各ストリームの深さは簡単なフロー解析によって求められる. ただし一般にすべてのプログラムに対してこのような d が求められるわけではない. たとえば, 以下のようなプログラムを考える.

```
while (...) x = x.a;
```

このプログラムは, `while` ループの実行回数に応じた深さの XML 文書を作るはずだが, この場合, アクション `x.a` が実行されるストリームの深さは一意には決まらない.

このようにあるストリーム変数の深さが一意に定まらないようなプログラムに対する推論は難しい. また (補間の問題を抜きにしても) このようにループをまわるたびに深さの増えるようなプログラムは次節でのべるような形ではコンパイルできない. しかし, 一方でこのようなプログラムはあまり現実的に価値はないだろうとわれわれは考える. なぜならば, 任意の深さの文書はループよりも再帰呼び出しによって作る方が自然であるからである (3.1 節の議論を参照のこと). そこで XP++ では上のようなプログラムは制約違反として除外することにした. この制約によって現実にプログラムが書きにくくなるという例は発見していない.

一般にわれわれは遷移系 $(S, \rightsquigarrow, S^{in}, s^{fin})$ およびアクション仕様 (α, d) が抽出できないようなプログラムはストリーム制約違反として除外する (2 節参照). 以降では仕様プログラムとはこの得られた遷移系およびアクション仕様のことであるとしてよい. また d に関して以下を仮定する.

仮定 1. 深さ d の作りかたによって一般に $s \rightsquigarrow t$ ならば $d(s) + 1 \geq d(t)$ である. また常に $\forall s \in S^{in}.\ d(s) = 1$ かつ $d(s^{fin}) = 0$ とする.

3.4 補間プログラム

補間プログラムとは XP++ プログラムのコンパイルによって得られる Java プログラムをモデル化したものである.

いま L を <σ>, </σ> からなる任意の文字列 λ の集合とする. 遷移系 $(S, \rightsquigarrow, S^{in}, s^{fin})$ が与えられたときアクション実現 (Q, β, q_0) は次のように定義される.

- 状態集合 Q
- 初期状態 $q_0\ (\in Q)$
- 遷移関数 β は $s \in S$ を受け取って, ある写像をかえす関数である. ただし $s \in S \setminus \{s^{fin}\}$ について $\beta(s) \in Q \to (L \times Q)$. また, s^{fin} について, $\beta(s^{fin}) \in Q \to L$ である.

補間プログラムは次のように実行される. いま対応する遷移系 $(S, \rightsquigarrow, S^{in}, s^{fin})$ の実行列

$$s_0\ (\in S^{in}) \rightsquigarrow \cdots \rightsquigarrow s_{n-1}\ (= s^{fin})$$

が与えられたとき, 遷移関数 β により以下のような出力列が求まる.

$$\begin{aligned}
&\beta(s_0)(q_0) = (\lambda_0, q_1),\\
&\beta(s_1)(q_1) = (\lambda_1, q_2),\\
&\quad\vdots\\
&\beta(s_{n-2})(q_{n-2}) = (\lambda_{n-2}, q_{n-1}),\\
&\beta(s_{n-1})(q_{n-1}) = \lambda_{n-1}
\end{aligned}$$

このとき

$$\beta(s_0 \cdots s_{n-1}, q_0) = \lambda_0 \cdots \lambda_{n-1}$$

と表記する. この $\lambda_0 \cdots \lambda_{n-1}$ が補間プログラムの出力結果である.

例 3. $Q = \{q_0, q_1, q_2, q_3, q_4\}, \beta$ が

$$\beta(s^{\texttt{xml.r}})(q_0) = (\texttt{<r>},\ q_1),$$
$$\beta(s^{\texttt{r.a}})(q_1) = (\texttt{<x><x></x><a>},\ q_2),$$
$$\beta(s^{\texttt{r.b(1)}})(q_1) = (\texttt{<x><x></x>},\ q_3),$$
$$\beta(s^{\texttt{r.a}})(q_2) = (\texttt{<a>},\ q_2),$$
$$\beta(s^{\texttt{r.b(1)}})(q_2) = (\texttt{},\ q_3),$$
$$\beta(s^{\texttt{r.b(2)}})(q_3) = (\texttt{</x>},\ q_4),$$
$$\beta(s^{fin})(q_4) = \texttt{</r>},$$

で定義されるとする. いま, 実行列

$$s^{\texttt{xml.r}} \rightsquigarrow s^{\texttt{r.b(1)}} \rightsquigarrow s^{\texttt{r.b(2)}} \rightsquigarrow s^{fin}$$

が与えられたとき, $q_0 \to q_1 \to q_3 \to q_4$ の順で β が実行され, 出力結果は次のようになる.

```
<r>
   <x>
      <x></x>
      <b></b>
   </x>
   <b></b>
</r>
```

このような補間プログラムを実際の Java プログラムに変換するのはそう難しくない. XP++ の処理系ではまず各ストリーム生成命令 `new` に対して状態変数を `int state = `q_0`;` のようにひとつ定義する. また各ノード生成命令に対して, これを `switch` 節でおきかえ, $\beta(s)$ (s はこのノード生成命令に対応する状態) を表現すればよい. たとえば $\beta(s)(q) = (\lambda, q')$ ならば,

```
switch(state) {
  case q:
    out.print(λ);
    state = q′;
    break;
  ···
}
```

のようになる.

3.5 仕様プログラムと補間プログラムの関係

遷移系 $(S, \rightsquigarrow, S^{in}, s^{fin})$, アクション仕様 (α, d) に対して, アクション実現 (Q, β, q_0) が与えられたとする. いま

$$\beta(s_0 \cdots s_{n-1}, q_0) = \lambda_0 \cdots \lambda_{n-1}$$

であったとする. このとき以下の $(i), (ii), (iii)$ の条件が必要となる.

(i) $\lambda_0 \cdots \lambda_{n-1}$ は開きタグ <σ> と閉じタグ </σ> のバランス (対応) のとれた XML 文書でなくてはならず, さらに指定された文法定義に合致していなくてはならない. すなわち,

$$\lambda_0 \cdots \lambda_{n-1} \in \mathcal{L}(\mathrm{G}^{in})$$

でなくてはならない.

(ii) 各 s_i について λ_i は α の指示する制約に合致した結果でなくてはならない. すなわち, $\alpha(s_i)$ = <σ> ならば $\lambda_i = \lambda'$<σ> でなくてはならない.

(iii) 各 s_i について λ_i は d で指定された開きタグと閉じタグの構造を守らなくてはならない. すなわち,

$$\lambda_0 \cdots \lambda_i (= \lambda' \texttt{<}\sigma\texttt{>}) \cdots \lambda_j (= \lambda'' \texttt{</}\sigma\texttt{>} \lambda''') \cdots \lambda_{n-1}$$

において $\alpha(s_i)$ = <σ> であるならば上に述べたように $\lambda_i = \lambda'$<σ> であるが, このとき開きタグ <σ> に対応する閉じタグ </σ> は

$$d(s_i) \geq d(s_j) \text{ かつ } i < j$$

であるような最小の j に関する出力文字列 λ_j の中に出現する.

条件 (iii) についてのみ補足する.

XP++では明示的に閉じタグを書かない. しかし 2 節で軽く触れたように, ストリーム `x` の上で `x.`σ によって σ-ノードを生成したときに, この σ-ノードをいつ閉じればいいかははっきりしている. すなわちそれは `x` またはその親ストリームのどれかに対して新しいノード生成が行われたときである. 深さ d はストリームの親子関係を示しているのだから, 条件 (iii) はまさにこのことを言っている.

さて, アクション実現 (Q, β, q_0) が任意のプログラム実行列 s_0 ($\in S^{in}$) $\rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_{n-1}$ ($= s^{fin}$) について, 上の条件 $(i), (ii), (iii)$ を満たすならば, (Q, β, q_0) は, (α, d) の実現であるという.

これで以上の定義から懸案のノード補間問題が次のように定式化できる.

定義 1. (ノード補間問題) 遷移系 $(S, \rightsquigarrow, s^{in}, s^{fin})$, それに対するアクション仕様 (α, d) およびプログラムの出力が満たすべき文法 $(\mathcal{G}, \to, \mathrm{G}^{in})$ が与えられたときに, アクション実現 (Q, β, q_0) で, (α, d) の実現になっているものがあるか調べよ. もしそのようなものがあればその (Q, β, q_0) のひとつを推論せよ.

3.6 問題解決のアイディア

本論文の補間プログラム推論のためのアルゴリズムの着想は余帰納的定義による非決定的オートマトンの包含の判定アルゴリズムから得た.

本研究でとりあつかうプログラムはオートマトンととてもよく似ている. プログラムは有限遷移系であり, 各プログラム点において, それぞれアクション (λ_i) が定義され, プログラムの全てのアクションの連結がプログラムの出力結果である.

次の節から具体的な補間プログラム推論アルゴリズムを紹介する. ただし読者の理解を助けるために以下にこのアルゴリズムのもとになった余帰納的定義による非決定的オートマトンの包含の判定アルゴリズムに触れる.

非決定的オートマトン $A = (\Sigma, Q_A, \to_A, I_A, F_A)$ と $B = (\Sigma, Q_B, \to_B, I_B, F_B)$ が与えられたとする. オートマトン A (B) の状態 s (t) に関する右言語 (= 状態 s あるいは状態 t から終了状態に到達する遷移で受理される語の集合) を $\mathcal{L}_A(s)$ ($\mathcal{L}_B(t)$) とする. 判定 $s \ \mathtt{sim}^{\uparrow}\ T$ (ただし $s \in Q_A, T \subseteq Q_B$) を

$$s \ \mathtt{sim}^{\uparrow}\ T \Leftrightarrow \mathcal{L}_A(s) \subseteq \bigcup_{t \in T} \mathcal{L}_B(t)$$

であるように定義したい. 特にこのとき

$$\mathcal{L}(A) \subseteq \mathcal{L}(B) \Leftrightarrow \forall s \in I_A.\ s \ \mathtt{sim}^{\uparrow}\ I_B$$

である.

この $s \ \mathtt{sim}^{\uparrow}\ T$ をどう定義するかを考える. $s \in F_A$ であるならば $\epsilon \in \mathcal{L}_A(s) \subseteq \bigcup_{t \in T} \mathcal{L}_B(t)$ であってほしいから, $F_B \cap T \neq \emptyset$ が必要である. また $s \overset{\sigma}{\to}_A s'$ であるような各 s' と σ について, $\mathcal{L}_A(s')$ が $\{w \mid \sigma w \in \bigcup_{t \in T} \mathcal{L}_B(t)\}$ $(= \bigcup_{\exists t \in T.\ t \overset{\sigma}{\to}_B t'} \mathcal{L}_B(t'))$ に含まれていればよい.

よって $s \ \mathtt{sim}^{\uparrow}\ T$ は次の規則を満たす最大の述語として定義できる. (余帰納法)

$$\frac{\begin{array}{c} s \in F_A \Rightarrow F_B \cap T \neq \emptyset, \\ \forall \sigma, s'.\ (s \overset{\sigma}{\to}_A s' \Rightarrow s' \ \mathtt{sim}^{\uparrow}\ \{t' \mid \exists t \in T.\ t \overset{\sigma}{\to}_B t'\}) \end{array}}{s \ \mathtt{sim}^{\uparrow}\ T}\ \textsc{Cind}$$

例 4. たとえば, $(\mathtt{a}, \mathtt{b})^* \subseteq (\mathtt{a} \cup \mathtt{b})^*$ は以下のように判定される. オートマトンは例えば以下のようである.

$$A = (\{\mathtt{a}, \mathtt{b}\}, \{s_0, s_1\}, \{s_0 \overset{\mathtt{a}}{\to} s_1,\ s_1 \overset{\mathtt{b}}{\to} s_0\}, \{s_0\}, \{s_0\}),$$
$$B = (\{\mathtt{a}, \mathtt{b}\}, \{t_0\}, \{t_0 \overset{\mathtt{a}}{\to} t_0,\ t_0 \overset{\mathtt{b}}{\to} t_0\}, \{t_0\}, \{t_0\})$$

判定は, 初期状態 $I_A = \{s_0\}$ および $I_B = \{t_0\}$ に関する $s_0 \ \mathtt{sim}^{\uparrow}\ \{t_0\}$ である. この $s_0 \ \mathtt{sim}^{\uparrow}\ \{t_0\}$ の導出木を下から上へと作っていくと以下のようになる.

$$\pi = \dfrac{\dfrac{\begin{array}{l} \vdots\ \pi \\ s_0 \ \mathtt{sim}^{\uparrow}\ \{t' \mid t_0 \overset{\mathtt{b}}{\to}_B t'\}\ (= \{t_0\}), \\ s_1 \overset{\mathtt{b}}{\to}_A s_0 \end{array}}{s_1 \ \mathtt{sim}^{\uparrow}\ \{t' \mid t_0 \overset{\mathtt{a}}{\to}_B t'\}\ (= \{t_0\}),}\ s_1 \notin F_A}{s_0 \overset{\mathtt{a}}{\to}_A s_1}\ F_B \cap \{t_0\} \neq \emptyset}{s_0 \ \mathtt{sim}^{\uparrow}\ \{t_0\}}$$

余帰納法による定義はループになっている導出も正しい導出として認めるという点で帰納法によるものと異なる. ここで $\overset{\pi}{\vdots}$ はこの $s_0 \ \mathtt{sim}^{\uparrow}\ \{t_0\}$ の導出がループになっており正しい導出であることを表現している.

いま上の判定 $s \ \mathtt{sim}^{\uparrow}\ T$ の双対の判定 $s \ \mathtt{sim}^{\downarrow}\ T$ をつぎのようなものとし,

$$s \ \mathtt{sim}^{\downarrow}\ T \Leftrightarrow \neg(s \ \mathtt{sim}^{\uparrow}\ \overline{T})$$

$s \ \mathtt{sim}^{\downarrow}\ \Omega$の抽象数えあげ (型検査)
↓
$s \ \mathtt{sim}^{\uparrow}\ \Omega \triangleright q, \lambda$を用いたプログラム推論
↓
ノード補間問題の解 (Q, β, q_0)

図 4: ノード補間問題 (定義 1) の解法

これを帰納的に定義することができる (ただし $\overline{T} = Q^B \setminus T$). この判定は以下のような規則を満たす最小の述語として定義できる [3] (帰納法).

$$\frac{s \in F_A,\ F_B \subseteq T}{s \ \mathtt{sim}^{\downarrow}\ T}\ \textsc{Base}$$

$$\frac{s \overset{\sigma}{\to}_A s',\ s' \ \mathtt{sim}^{\downarrow}\ T',\ \{t \mid \exists t' \in T'.\ t \overset{\sigma}{\to}_B t'\} \subseteq T}{s \ \mathtt{sim}^{\downarrow}\ T}\ \textsc{Ind}$$

包含の判定は次のようになる.

$$\mathcal{L}(A) \subseteq \mathcal{L}(B) \Leftrightarrow \forall s \in I_A.\ \forall T.\ (s \ \mathtt{sim}^{\downarrow}\ T \Rightarrow T \cap I_B \neq \emptyset)$$

この包含判定を行うためには $s \ \mathtt{sim}^{\downarrow}\ T$ を満たす全ての s と T の組を数えあげなくてはならないことに注意する. これを求めるためには, 規則 Base から開始し, 到達可能な $s \ \mathtt{sim}^{\downarrow}\ T$ を規則 Ind によって全て数えあげる以外に手はない. 余帰納法による包含アルゴリズムは $s_0 \ \mathtt{sim}^{\uparrow}\ \{t_0\}$ の導出木を下から上へと作っていく推論である. 一方, 帰納法にもとづく場合, アルゴリズムは上から下への数えあげによるものとなる.

さて, ここで 2 つのオートマトンの包含のアルゴリズムを説明してきたのはこの両者の関係が, これから補間プログラム推論アルゴリズムを説明するときの鍵となる関係だからである. 実際, 次節から述べる技法はこの推論と数えあげを組み合わせた巧妙なものとなっている.

4 問題の解法

4.1 プログラム推論

本題に入る前に, 文法 $(\mathcal{G}, \to, \mathrm{G}^{in})$ についての若干の補足をする.

いま Ω で終端記号, 非終端記号からなる文字列 ω の集合をあらわす. 与えられた XML 文書が文法 $(\mathcal{G}, \to, \mathrm{G}^{in})$ に合致するかどうかをこのような Ω を用いて検査するアルゴリズムが存在する.

[3] 機械的な書き換えでは, Ind の前提が, ある T' および T'' に対して, $....s' \ \mathtt{sim}^{\downarrow}\ T'', T'' \subseteq T', ...$ という形になった定義が得られる. しかし作られた判定を調べると「T'' が存在し $s' \ \mathtt{sim}^{\downarrow}\ T''$ かつ $T'' \subseteq T'$」であるときまたそのときにかぎり $s' \ \mathtt{sim}^{\downarrow}\ T'$ が成立するので, Ind のように定義してよい.

$$\pi_0 = \dfrac{\dfrac{}{\mathrm{G}^{in} \xrightarrow{\texttt{<r>}} \mathrm{G}^{\texttt{x}|}\eta \rhd \epsilon, \texttt{</r>}}\ \textsc{Inf-push0} \qquad \theta_1 : \ \begin{array}{c}\pi_1 \\ \vdots \\ s^{\texttt{r.a}}\ \texttt{sim}^{\uparrow}\ \{\mathrm{G}^{\texttt{x}|}\eta\} \rhd q_1, \texttt{</r>}\end{array} \qquad \begin{array}{c}\pi'_1 \\ \vdots \\ s^{\texttt{r.b(1)}}\ \texttt{sim}^{\uparrow}\ \{\mathrm{G}^{\texttt{x}|}\eta\} \rhd q_1, \texttt{</r>}\end{array}}{\theta_0 : s^{\texttt{xml.r}}\ \texttt{sim}^{\uparrow}\ \{\mathrm{G}^{in}\} \rhd q_0, \epsilon}\ \textsc{Inf-cind}$$

$$\pi_1 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{<a>}} \eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</x></r>}, \texttt{</x></r>}}\ \textsc{Inf-push0}}{\eta\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{</x><a>}} \eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</x></x></r>}, \texttt{</x></r>}}\ \textsc{Inf-pop}}{\mathrm{G}^{\texttt{x}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{<x></x><a>}} \eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</x></r>}, \texttt{</x></r>}}\ \textsc{Inf-push}}{\mathrm{G}^{\texttt{x}|}\eta \xrightarrow{\texttt{<x><x></x><a>}} \eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</r>}, \texttt{</x></r>}}\ \textsc{Inf-push} \qquad \theta_2 : \ \begin{array}{c}\pi_2 \\ \vdots \\ s^{\texttt{r.a}}\ \texttt{sim}^{\uparrow}\ \{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta\} \\ \rhd q_2, \texttt{</x></r>}\end{array} \qquad \begin{array}{c}\pi'_2 \\ \vdots \\ s^{\texttt{r.b(1)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta\} \\ \rhd q_2, \texttt{</x></r>}\end{array}}{\theta_1 : s^{\texttt{r.a}}\ \texttt{sim}^{\uparrow}\ \{\mathrm{G}^{\texttt{x}|}\eta\} \rhd q_1, \texttt{</r>}}\ \textsc{Inf-cind}$$

$$\pi'_1 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{}} \cdots \rhd \texttt{</x></r>}, \cdots}\ \textsc{Inf-push0}}{\eta\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{</x>}} \cdots \rhd \texttt{</x></x></r>}, \cdots}\ \textsc{Inf-pop}}{\mathrm{G}^{\texttt{x}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{<x></x>}} \cdots \rhd \texttt{</x></r>}, \cdots}\ \textsc{Inf-push}}{\mathrm{G}^{\texttt{x}|}\eta \xrightarrow{\texttt{<x><x></x>}} \eta|\eta\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</r>}, \texttt{</x></r>}}\ \textsc{Inf-push} \qquad \begin{array}{c}\pi_3 \\ \vdots \\ s^{\texttt{r.b(2)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta\} \\ \rhd q_3, \texttt{</x></r>}\end{array}}{s^{\texttt{r.b(1)}}\ \texttt{sim}^{\uparrow}\ \{\mathrm{G}^{\texttt{x}|}\eta\} \rhd q_1, \texttt{</r>}}\ \textsc{Inf-cind}$$

$$\pi_2 = \dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{<a>}} \cdots \rhd \texttt{</x></r>}, \cdots}\ \textsc{Inf-push0}}{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{<a>}} \eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</x></r>}, \texttt{</x></r>}}\ \textsc{Inf-pop} \qquad \begin{array}{c}\pi_2 \\ \vdots \\ s^{\texttt{r.a}}\ \texttt{sim}^{\uparrow}\ \{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta\} \\ \rhd q_2, \texttt{</x></r>}\end{array} \qquad \theta_3 : \ \begin{array}{c}\pi'_2 \\ \vdots \\ s^{\texttt{r.b(1)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta\} \\ \rhd q_2, \texttt{</x></r>}\end{array}}{\theta_2 : s^{\texttt{r.a}}\ \texttt{sim}^{\uparrow}\ \{\eta|\mathrm{G}^{\texttt{ab}}\mathrm{G}^{\texttt{ab}|}\eta\} \rhd q_2, \texttt{</x></r>}}\ \textsc{Inf-cind}$$

$$\pi'_2 = \dfrac{\dfrac{\dfrac{}{\eta\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{}} \cdots \rhd \texttt{</x></r>}, \cdots}\ \textsc{Inf-push0}}{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{}} \eta|\eta\mathrm{G}^{\texttt{ab}|}\eta \rhd \texttt{</x></r>}, \texttt{</x></r>}}\ \textsc{Inf-pop} \qquad \theta_4 : \ \begin{array}{c}\pi_3 \\ \vdots \\ s^{\texttt{r.b(2)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta\} \rhd q_3, \texttt{</x></r>}\end{array}}{\theta_3 : s^{\texttt{r.b(1)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta\} \rhd q_2, \texttt{</x></r>}}\ \textsc{Inf-cind}$$

$$\pi_3 = \dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{}} \cdots \rhd \texttt{</r>}, \cdots}\ \textsc{inf-push0}}{\eta\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{</x>}} \cdots \rhd \texttt{</x></r>}, \cdots}\ \textsc{Inf-pop}}{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta \xrightarrow{\texttt{</x>}} \eta|\eta|\eta \rhd \texttt{</x></r>}, \texttt{</r>}}\ \textsc{Inf-pop} \qquad \theta_5 : \ \begin{array}{c}\pi_4 \\ \vdots \\ s^{fin}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta|\eta\} \\ \rhd q_4, \texttt{</r>}\end{array}}{\theta_4 : s^{\texttt{r.b(2)}}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta\mathrm{G}^{\texttt{ab}|}\eta\} \rhd q_3, \texttt{</x></r>}}\ \textsc{Inf-cind}$$

$$\pi_4 = \dfrac{\dfrac{\dfrac{\dfrac{}{\eta \xrightarrow{\epsilon} \eta \rhd \epsilon, \epsilon}\ \textsc{Inf-pop0}}{\eta|\eta \xrightarrow{\texttt{</r>}} \eta \rhd \texttt{</r>}, \epsilon}\ \textsc{Inf-pop}}{\eta|\eta|\eta \xrightarrow{\texttt{</r>}} \eta \rhd \texttt{</r>}, \epsilon}\ \textsc{Inf-pop}}{\theta_5 : s^{fin}\ \texttt{sim}^{\uparrow}\ \{\eta|\eta|\eta\} \rhd q_4, \texttt{</r>}}\ \textsc{Inf-base}$$

図 5: 導出の例 (ラベル $\theta_0, \ldots, \theta_5$ は証明で用いるパス θ の例)

たとえば 3.2 節の文法と XML 文書を例にとる. このアルゴリズムは以下のように動く.

	$\{\mathrm{G}^{in}\}$ $(= \Omega^{in})$
<r>	$\{\mathrm{G}^{\mathrm{x}}\eta\}$
<x>	$\{\mathrm{G}^{\mathrm{x}}\mathrm{G}^{\mathrm{ab}}\eta, \eta\mathrm{G}^{\mathrm{ab}}\eta\}$
<x></x>	$\{\mathrm{G}^{\mathrm{ab}}\mathrm{G}^{\mathrm{ab}}\eta\}$
<a>	$\{\mathrm{G}^{\mathrm{ab}}\mathrm{G}^{\mathrm{ab}}\eta\}$
<a>	$\{\mathrm{G}^{\mathrm{ab}}\mathrm{G}^{\mathrm{ab}}\eta\}$
	$\{\eta\mathrm{G}^{\mathrm{ab}}\eta\}$
</x>	$\{\mathrm{G}^{\mathrm{ab}}\eta\}$
	$\{\eta\eta\}$
</r>	$\{\eta\}$ $(= \Omega^{fin})$

つまり記号列の集合 Ω を記憶しながら, 文書を上から下にむかって走査すればよいのである. Ω は生成規則 $\rightarrow$ を用いて遷移させる. たとえば $\mathrm{G}^{\mathrm{x}} \rightarrow$ <x>G^{x}</x>G^{ab} ならば <x> を読んだとき, Ω 中の $\mathrm{G}^{\mathrm{x}}\omega$ を $\mathrm{G}^{\mathrm{ab}}\mathrm{G}^{\mathrm{x}}\omega$ に変えるように遷移する. また </σ> を読んだときはいつでも $\eta\omega \in \Omega$ であるような $\eta\omega$ から先頭の η を取り除いた列の集合に遷移する. $\Omega^{in} = \{\mathrm{G}^{in}\}$ から開始し, $\eta \in \Omega^{fin}$ であるようなある Ω^{fin} に至ることができればこの XML 文書は文法に合致する.

以上のアルゴリズムを理解したうえで, まず次のことを調べるための判定について議論する.

- 入力: 遷移系 $(S, \rightsquigarrow, s^{in}, s^{fin})$, アクション実現 (Q, β, q_0), アクション仕様 (α, d)
- 出力: (Q, β, q_0) が実際に (α, d) の実現になっているかどうか.

判定 $s\ \mathtt{sim}^{\uparrow}\ \Omega \triangleright q, \lambda^c$ は, おおまかにいって「s から始まる任意の補間プログラムの実行結果を, 上記アルゴリズムを用いて走査したとき, Ω から始めれば $\eta \in \Omega^{fin}$ であるような Ω^{fin} に至ることができる」という意味を持つ.

この判定は以下の規則を満たす最大の述語として定義される. ただし, ここでは Ω は終端記号, 非終端記号に加えて区切り「|」を含むような文字列の集合である. この区切り「|」は条件 (*iii*) を満たすかどうかのためだけに必要であり, そう重要ではない. また実際には s の出力結果は Ω だけではなく, λ^c に合致しなくてはならない. すなわち λ^c は </σ> だけからなる文字列であり, この順で閉じタグが出力されなくてはならないことを示す.

$$\frac{\beta(s^{fin})(q) = \lambda,\quad \exists\omega \in \Omega.\ \omega \xrightarrow{\lambda} \eta \triangleright \epsilon, \epsilon}{s^{fin}\ \mathtt{sim}^{\uparrow}\ \Omega \triangleright q, \epsilon}\ \textsc{Inf-base}$$

$$\frac{\begin{array}{c}\beta(s)(p) = (\lambda\text{<}\sigma\text{>}, q),\quad \alpha(s) = \text{<}\sigma\text{>},\\ \forall\omega \in \Omega.\ \omega\text{内の}|\text{の数} = d(s),\\ \Omega' = \{\omega' \mid \exists\omega \in \Omega.\ \omega \xrightarrow{\lambda\text{<}\sigma\text{>}} \omega' \triangleright \lambda^c, \lambda^{c\prime}\},\\ \forall t.\ s \rightsquigarrow t \Rightarrow t\ \mathtt{sim}^{\uparrow}\ \Omega' \triangleright q, \lambda^{c\prime}\end{array}}{s\ \mathtt{sim}^{\uparrow}\ \Omega \triangleright p, \lambda^c}\ \textsc{Inf-cind}$$

以上の定義のポイントは $\forall t.s \rightsquigarrow t \Rightarrow$ .... の部分であり, ここで s と Ω に関する問題が, プログラムの各分岐先 t とそして Ω の遷移先である Ω' に関する部分問題に分解される. いっぽう, 実際に本節冒頭のアルゴリズムを使い, アクション λ<σ> が Ω から Ω' への遷移に対応するかどうかを検査しているのは, 定義で用いられている判定 $\omega \xrightarrow{\lambda\text{<}\sigma\text{>}} \omega' \triangleright \lambda^c, \lambda^{c\prime}$ である.

この判定 $\omega \xrightarrow{\lambda^o} \omega' \triangleright \lambda^c, \lambda^{c\prime}$ は以下のように定義され, λ^o を走査するときに ω から ω' に遷移することができるということを意味する. ただし, $\lambda^c, \lambda^{c\prime}$ は閉じタグからなる文字列であり, これから閉じなければならないタグの列を示す. λ^o, λ^c は入力, $\lambda^{c\prime}$ は出力とみなせる.

$$\frac{\mathrm{G} \rightarrow \text{<}\sigma\text{>E</}\sigma\text{>F}}{\mathrm{G}\omega \xrightarrow{\text{<}\sigma\text{>}} \mathrm{E}|\mathrm{F}\omega \triangleright \lambda^c, \text{</}\sigma\text{>}\lambda^c}\ \textsc{Inf-push0}$$

$$\frac{\begin{array}{c}\mathrm{G} \rightarrow \text{<}\sigma\text{>E</}\sigma\text{>F},\\ \mathrm{EF}\omega \xrightarrow{\lambda^o} \omega' \triangleright \text{</}\sigma\text{>}\lambda^c, \lambda^{c\prime}\end{array}}{\mathrm{G}\omega \xrightarrow{\text{<}\sigma\text{>}\lambda^o} \omega' \triangleright \lambda^c, \lambda^{c\prime}}\ \textsc{Inf-push}$$

$$\frac{}{\eta \xrightarrow{\epsilon} \eta \triangleright \epsilon, \epsilon}\ \textsc{Inf-pop0}$$

$$\frac{\begin{array}{c}\omega = \eta|\omega' \vee \omega = \eta\omega',\\ \omega' \xrightarrow{\lambda^o} \omega'' \triangleright \lambda^c, \lambda^{c\prime}\end{array}}{\omega \xrightarrow{\text{</}\sigma\text{>}\lambda^o} \omega'' \triangleright \text{</}\sigma\text{>}\lambda^c, \lambda^{c\prime}}\ \textsc{Inf-pop}$$

さて以上の定義に関して次がいえる.

定理 1. すべての $s_0 \in S^{in}$ について $s_0\ \mathtt{sim}^{\uparrow}\ \{\mathrm{G}^{in}\} \triangleright q_0, \epsilon$ であれば, プログラムの任意の実行列は *3.5* 節の (*i*), (*ii*), (*iii*) の条件を満たす.

証明スケッチ. いま, 以下のようなプログラムの実行列を仮定する.

$$s_0\ (\in S^{in}) \rightsquigarrow \cdots \rightsquigarrow s_{n-1}\ (= s^{fin})$$

もし, この実行列が定理の条件を満たすならば, すなわち任意の実行列が定理の条件を満たす. さて証明はいま仮定された実行列と $s_0\ \mathtt{sim}^{\uparrow}\ \{\mathrm{G}^{in}\} \triangleright q_0, \epsilon$ の導出木上のある導出パスを比較することによる.

実際, Inf-cind のルールをよくみてみると, 導出木はこの規則を用いて各 s_i に対して $s_i \rightsquigarrow s_{i+1}$ であるような全ての s_{i+1} へ分岐することがわかる. いいかえると導出木上の $s_0\ \mathtt{sim}^{\uparrow}\ \{\mathrm{G}^{in}\} \triangleright q_0, \epsilon$ から $s^{fin}\ \mathtt{sim}^{\uparrow}\ \Omega \triangleright q_{n-1}, \epsilon$ に至るパス θ で, パラメータとして出現する s の列が与えられたプログラムの実行列と等しいものが存在する.

ここでは (*ii*), (*iii*) の条件に関する証明は略し ((*iii*) の証明は 3.3 節でのべた仮定 1 を使う), (*i*) の条件についてのみを証明する. たとえば 3.3 節の例で与えたプログラムに対して

$$\begin{array}{l} s^{\mathtt{xml.r}}(= s_0) \rightsquigarrow s^{\mathrm{r.a}}(= s_1) \rightsquigarrow s^{\mathrm{r.a}}(= s_2)\\ \rightsquigarrow s^{\mathrm{r.b(1)}}(= s_3) \rightsquigarrow s^{\mathrm{r.b(2)}}(= s_4) \rightsquigarrow s^{fin}(= s_5)\end{array}$$

のような実行列が仮定されたとする. このときは導出木上のパス θ は図 5 に番号で示したようなものとなる. さらに (Q,β,q_0) が 3.4 節の例のようだったとする. 補間プログラムの実行は

$$\begin{aligned}
&\beta(s_0,q_0) = (\texttt{<r>}(=\lambda_0), q_1),\\
&\beta(s_1,q_1) = (\texttt{<x><x></x><a>}(=\lambda_1), q_2)\\
&\beta(s_2,q_2) = (\texttt{<a>}(=\lambda_2), q_2),\\
&\beta(s_3,q_2) = (\texttt{}(=\lambda_3), q_3),\\
&\beta(s_4,q_3) = (\texttt{</x>}(=\lambda_4), q_4)\\
&\beta(s_5,q_4) = \texttt{</r>}(=\lambda_5),
\end{aligned}$$

のようになる.

このように計算された列 $\lambda_0\cdots\lambda_{n-1}$ に対して

$$\lambda_0\cdots\lambda_{n-1} \in \mathcal{L}(\mathrm{G}^{in})$$

を証明したい. 証明はこの $\lambda_0\cdots\lambda_{n-1}$ の長さに関する帰納法による. 本節の冒頭の議論をふまえ, さらに各 θ_i の side condition に存在する $\omega \xrightarrow{\lambda^o} \omega' \triangleright \lambda^c, \lambda^{c\prime}$ の導出に着目すればほぼ明らか. たとえば θ_1 から θ_2 に辿るためには,

$$\mathrm{G}^{\mathtt{x}|}\eta \xrightarrow{\texttt{<x><x></x><a>}} \omega' \triangleright \texttt{</r>}, \texttt{</x></r>}$$

を満たす ω' を数えることが必要だが, この ω' を求める途中で $\rightarrow$ の左辺に現われる列 $\mathrm{G}^{\mathtt{x}|}\eta, ..., \mathrm{G}^{\mathtt{ab}}\mathrm{G}^{\mathtt{ab}|}\eta$ は本節冒頭の例でみたものと区切り $|$ をのぞき同じであることに注意する. □

さらに, $s\,\mathtt{sim}^{\uparrow}\,\Omega \triangleright q, \lambda^c$ はアクション実現 (Q,β,q_0) の推論アルゴリズムとして考えることもできる. このときは, Inf-cind の side condition である $\Omega' = \{\omega' \mid \exists\omega \in \Omega.\ \omega \xrightarrow{\lambda\texttt{<}\sigma\texttt{>}} \omega' \triangleright \lambda^c, \lambda^{c\prime}\}$ 部分は β を定義するための適当な λ<σ> を探しだすアルゴリズムだと考えることができる. また, 状態 q についてはもし有限の導出木を作れるならば, たかだかそのノードの数が必要であるだけである. とはいえ, この推論アルゴリズムは一般には止まらない. どうやってこの推論アルゴリズムを止めるかについては次節で議論する.

4.2 判定 $\mathtt{sim}^{\downarrow}$ の数えあげ

前節で述べた $s\,\mathtt{sim}^{\uparrow}\,\Omega \triangleright q, \lambda^c$ を用いた補間プログラム推論のアルゴリズムはこれだけでは使いものにならない. 理由は (1) 停止するとは限らない (2) バックトラックが頻繁に起こり効率が悪いという 2 点である. なぜそうなるかというと, Inf-cind の side condition である

$$\Omega' = \{\omega' \mid \exists\omega \in \Omega.\ \omega \xrightarrow{\lambda\texttt{<}\sigma\texttt{>}} \omega' \triangleright \lambda^c, \lambda^{c\prime}\},$$

がさまざまな λ<σ> と Ω' のペアを捜し出してくるからであり, ナイーブな手法ではいったいそのうちどの λ<σ> と Ω' を使って導出木を構築していくのが正しいかわからないためである. たとえば図 5 の π_1 の導出においても, <x></x><a> ではその後の導出木の構築がうまくいかない. そこで <x><x></x><a> を推論しなくてはならない. そもそもこの部分が求める Ω' は一般に有限ではない.

解決法はいくつかある. たとえばバックトラックが起こったら計算しなおさずに型エラーとする. という方法もある. 確かに図 5 のようなプログラムの後方に依存するようなノード補間の推論がそれほど重要だとは思われない部分もある.

しかしわれわれは別の解決策を探した. 結果的にわれわれが取った手法はまずプログラムの後方から前方に向けての検査を行い, しかるのちに, そこで得られた情報を用いて, プログラムの前方から後方への推論を行うという 2 パスによる方法である (図 4). プログラムの後方から得られた情報を用いれば Inf-cind をどこで実行すればいいかをバックトラックなしで決定することができる.

定理 2. いま $s\,\mathtt{sim}^{\uparrow}\,\Omega$ は「ある (Q,β,q_0) および $q \in Q$, λ^c が存在し, $s\,\mathtt{sim}^{\uparrow}\,\Omega \triangleright q, \lambda^c$ を満たす」という意味とする. すると

$$s\,\mathtt{sim}^{\uparrow}\,\Omega \Leftrightarrow \forall\Omega'.\ (s\,\mathtt{sim}^{\downarrow}\,\Omega' \Rightarrow \Omega \cap \Omega' \neq \emptyset)$$

という性質を満たす判定 $s\,\mathtt{sim}^{\downarrow}\,\Omega$ を, 帰納的に定義することができる.

証明スケッチ. 判定 $s\,\mathtt{sim}^{\uparrow}\,\Omega$ は図 6 のように定義できる (余帰納法). いまこの判定の導出木 π があったとする. すると適当な (Q,β,q_0) を定義し, これに対して, π と同じ形の $s\,\mathtt{sim}^{\uparrow}\,\Omega \triangleright q.\lambda_c$ の導出木 π' を作ることができる. また $s\,\mathtt{sim}^{\uparrow}\,\Omega \triangleright q.\lambda_c$ の導出木 π' から β に関する条件部を忘れて, $s\,\mathtt{sim}^{\uparrow}\,\Omega$ の導出木 π が作れる.

さらに 3.6 節 で触れたように, 余帰納的定義を双対の関係にある帰納的定義に書き換えることで,

$$s\,\mathtt{sim}^{\uparrow}\,\Omega \Leftrightarrow \neg s(\mathtt{sim}^{\downarrow}\overline{\Omega})$$

であるような $\mathtt{sim}^{\downarrow}$ を定義できる. これは, 図 7 のようになる (帰納法). さて, この定義をみると明らかにこの判定は Ω に関して upward closed つまり, $s\,\mathtt{sim}^{\downarrow}\,\Omega \wedge \Omega \subseteq \Omega' \Rightarrow s\,\mathtt{sim}^{\downarrow}\,\Omega'$ である. よって,

$$\begin{aligned}
&s\,\mathtt{sim}^{\uparrow}\,\Omega\\
&\Leftrightarrow \neg(s\,\mathtt{sim}^{\downarrow}\,\overline{\Omega})\\
&\Leftrightarrow \nexists\Omega'.\ \Omega' \subseteq \overline{\Omega} \wedge s\,\mathtt{sim}^{\downarrow}\,\Omega'\\
&\Leftrightarrow \forall\Omega'.\ (s\,\mathtt{sim}^{\downarrow}\,\Omega' \Rightarrow \Omega \cap \Omega' \neq \emptyset)
\end{aligned}$$

□

特にこの $\mathtt{sim}^{\downarrow}$ を使って, 型検査が可能である. このためには以下の性質を調べればよい.

系 1.

$$\begin{aligned}
&\forall s \in S^{in}.\ (s\,\mathtt{sim}^{\downarrow}\,\Omega \Rightarrow \mathrm{G}^{in} \in \Omega)\\
&\Leftrightarrow\\
&\text{アクション実現 } (Q,\beta,q_0) \text{ が存在して,}\\
&\qquad \forall s \in S^{in}.\ s\,\mathtt{sim}^{\uparrow}\,\{\mathrm{G}^{in}\} \triangleright q_0, \epsilon
\end{aligned}$$

$$\frac{\exists \omega \in \Omega, \lambda.\ \omega \xrightarrow{\lambda} \eta}{s^{fin}\ \mathtt{sim}^{\uparrow}\ \Omega}\ \textsc{Cenum-base} \qquad \frac{\begin{array}{c}\exists \lambda.\ \Omega' = \{\omega' \mid \exists \omega \in \Omega.\ \omega \xrightarrow{\lambda\alpha(s)} \omega'\},\\ \forall \omega \in \Omega'.\ \omega\text{内の}\mid\text{の数} = d(s),\\ \forall t.\ s \rightsquigarrow t \Rightarrow t\ \mathtt{sim}^{\uparrow}\ \Omega'\end{array}}{s\ \mathtt{sim}^{\uparrow}\ \Omega}\ \textsc{Cenum-cind}$$

$$\frac{\text{G} \rightarrow \text{<}\sigma\text{>E</}\sigma\text{>F}}{\text{G}\omega \xrightarrow{\text{<}\sigma\text{>}} \text{E}^{\mid}\text{F}\omega}\ \textsc{Enum-push0} \qquad \frac{\begin{array}{c}\text{G} \rightarrow \text{<}\sigma\text{>E</}\sigma\text{>F},\\ \text{EF}\omega \xrightarrow{\lambda} \omega'\end{array}}{\text{G}\omega \xrightarrow{\text{<}\sigma\text{>}\lambda} \omega'}\ \textsc{Enum-push}$$

$$\frac{}{\eta \xrightarrow{\epsilon} \eta}\ \textsc{Enum-pop0} \qquad \frac{\begin{array}{c}\omega = \eta^{\mid}\omega' \vee \omega = \eta\omega',\\ \omega' \xrightarrow{\lambda} \omega''\end{array}}{\omega \xrightarrow{\text{</}\sigma\text{>}\lambda} \omega''}\ \textsc{Enum-pop}$$

図 6: 判定 $s\ \mathtt{sim}^{\uparrow}\ \Omega$ の定義 (余帰納的定義)

$$\frac{\{\omega \mid \omega \xrightarrow{\lambda} \eta\} \subseteq \Omega}{s^{fin}\ \mathtt{sim}^{\downarrow}\ \Omega}\ \textsc{Enum-base} \qquad \frac{\begin{array}{c}\forall \lambda.\ \exists t.\ \exists \Omega'.\ s \rightsquigarrow t,\ t\ \mathtt{sim}^{\downarrow}\ \Omega',\\ \left\{\begin{array}{l}\omega \mid\\ \exists \omega' \in \Omega'.\\ \omega'\text{内の}\mid\text{の数} = d(s),\\ \omega \xrightarrow{\lambda\alpha(s)} \omega'\end{array}\right\} \subseteq \Omega\end{array}}{s\ \mathtt{sim}^{\downarrow}\ \Omega}\ \textsc{Enum-ind}$$

図 7: 数えあげ $s\ \mathtt{sim}^{\downarrow}\ \Omega$ の定義 (帰納的定義)

われわれの XP++ の処理系ではこれを調べることによってプログラム推論の答えが存在しない可能性がある場合 (型エラー) にはもとから推論を行わない. これによって推論アルゴリズムが停止しない問題を防ぐことができる. さらに数えあげを用いて, 4.1 節の Inf-cind の規則を用いるのを,

$$\forall \Omega''.\ (t\ \mathtt{sim}^{\downarrow}\ \Omega'' \Rightarrow \Omega' \cap \Omega'' \neq \emptyset)$$

が成立する場合のみに限定できる.

さて, 実は判定 $s\ \mathtt{sim}^{\downarrow}\ \Omega$ じたいの数えあげは実はまだ, 簡単ではない. なぜならば Ω の可能性は有限ではないからである. しかし我々は $s\ \mathtt{sim}^{\downarrow}\ \Omega$ を満たす Ω の集合をぴったり数える必要はない. そうではなく この Ω の集合よりも大きいなんらかの集合をどうにかして数えあげればよい. このために抽象数えあげの手法を使う. たとえばこのためには, Ω をある定数 k 以上の長さのもの Ω_0 とそれ以上の Ω_1 にわけ (Ω は Ω_0 と Ω_1 の disjoint union), $\text{G}_0 \ldots \text{G}_{k-1}\omega \in \Omega_1$ については $\text{G}_0 \ldots \text{G}_{k-1}$ のみを覚えておく, という手法が考えられる. われわれは当初この方法も試したが, 実用上はあまり効率よくなかった. そこで実装ではもう少し効率のよい方法で抽象数えあげを実現しようとしているが, すこし繁雑であり, またよりよい手法を模索中なのでここではこれ以上触れない.

5 まとめと課題

型つき XML 処理言語 XP++ の言語の仕様と, そのノード補間の技法について概説した. この言語自体の課題としてはどのように実用性をましていくかという点につきる. ストリーム処理に対する型検査という発想自体はよいと思われる. 実用性という面からみると, 新しい XML 処理言語およびコンパイラよりも, 独立した Java の API でしかも型検査ができるものという技術の方がよいのではないかと思われる部分もあり, この方向での再デザインも検討している.

またノード補間の技法については, もう少し簡単な手法はないかを模索している. また, 今回あつかった XP++ はもちろん Turing 完全な命令言語で, よって完全なノード補間問題の検査は不可能である. しかし, 抽出された仕様プログラムだけを考えたときのノード補間問題が決定可能なのかどうなのか, そもそもまだはっきりわかっていない. 現在のところは完全な解はあきらめ近似解を用いている. これについてもうすこしきちんと答えを出す必要がある.

6 謝辞

本研究を進める際の, 村田真氏の助言に感謝します. またソフトウェアの公開を手伝ってくださった IBM alphaWorks の関係者の方々にも感謝します. また詳しくみてくださり, 多くの有用な指摘, コメントをくださった査読者のかたがたに感謝します.

参考文献

[1] I. Avila-Campillo, T. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable xml stream processing. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002.

[2] C. Barton, P. Charles, M. Fontoura, D. Goyal, V. Josifovski, and M. Raghavachari. An algorithm for streaming XPath processing with forward and backward axes. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002.

[3] A. S. Christensen, A. Muller, and M. I. Schwartzbach. Static analysis for dynamic XML. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002.

[4] J. Coelho and M. Florido. Type-based XML processing in logic programming. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, 2003.

[5] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[6] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proceedings of 3rd Intl. Workshop on the Web and Databases (WebDB)*, 2000.

[7] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Proceedings of Extreme Markup Language 2001, Montreal*, pages 153–166, 2001.

[8] K. Nakano and S. Nishimura. Deriving event-based document transformers from tree-based specifications. In *LDTA'2001 Workshop on Language Descriptions, Tools and Applications*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. available on-line: http://www.elsevier.nl/gej-ng/31/29/23/73/27/show/Products/notes/index%.htt.

[9] Oraganization for Advancement of Structured Information Standards (OASIS). RELAX NG, 2001. http://www.oasis-open.org/committees/relax-ng/.

[10] B. C. Pierce, et al. Xtatic project, 2002. http://ww.cis.upenn.edu/~bcpierce/xtatic.

[11] World Wide Web Consortium. XML query algebra, 2000. http://www.w3.org/TR/query-algebra/.