XQuery 問合せのある最適化手法における対象クラスの拡張

2019SC045 新美涼香　2019SC062 田中景大

指導教員：石原靖哲

1　はじめに

近年，インターネット上では大量のデータがやりとりされており，データを記述する言語として XML（Extensible Markup Language）が広く利用されている．XML とは，構造化データを記述可能なマークアップ言語のひとつである．XML 文書は木構造を持ち，木構造を構成する要素をノードという．また，XML 文書に対してノードの探索・抽出などの問合せを行う関数型言語として XQuery が広く用いられている．

XML 文書はその構造上，ノードの順序が重要であり，下方向以外の探索は多数の重複を発生させることから，XQuery は DDO（Distinct Document Order）処理が実行されるように設計されている．DDO 処理とは，入力文書を処理する際に，中間結果として得られたノード系列を文書順に並び替え，重複するノードを削除する処理である．しかし，中間結果として得られるノード系列が大きくなると，DDO 処理に要する時間と記憶容量が無視できなくなるという問題がある．そこで石原ら [2] は，与えられた XQuery 式を，DDO 処理を含まない等価な式に変換する手法を提案している．この手法によって，変換後の式が元の式より時間と記憶容量を節約できる場合があることを確認している．しかし，石原らの手法では対象クラスが限られているという課題があった．

本研究では，石原らの手法で let 構文と element constructor 構文（以下 ec 構文）を扱えるように新たに変換手順や変換規則を加えることで，対象クラスの拡張を行う．let 構文と ec 構文が扱えるようになることで，探索結果を利用した新しい XML 文書の生成が可能となり，ノードの要素名と値を動的に設定できるという利点がある．また，提案手法に従って変換した XQuery 式と元の式を対象に，実行時間と使用メモリ量の比較実験を行い，変換後の式が実行時間を節約できる場合があることを確認した．

2　DDO 処理の問題点とその解決策

前述したように，DDO 処理とは入力文書を処理する際に，中間結果として得られたノード系列を並び替え，重複するノードを削除する処理である．DDO 処理は XML 文書内でノードを探索する際に行われる．より具体的には，XQuery 式のスラッシュ '/' が現れる箇所で DDO 処理が発生する．ここで，図 2 に示す XQuery 式 e_{in} で行われる DDO 処理について例 1 で説明する．

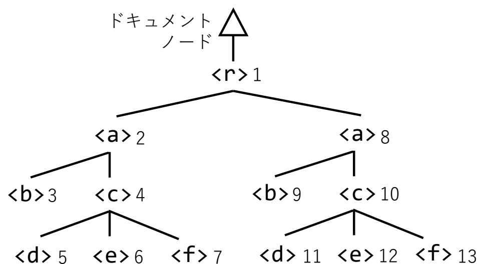

図 1　XML 文書 input.xml の構造

例 1　XQuery 式 e_{in} を，図 1 に示す XML 文書 input.xml に対して評価する状況を考える．なお，図 1 の各ノードには文書順に番号が振られており，この番号によってノードを示す．また，1 行目の for 構文で`$a` に 3 が束縛されたときに ec 構文で生成される木（以下 ec 木）のノードを指す番号には $'$（プライム），`$a` に 9 が束縛されたときに生成される ec 木のノードを指す番号には $''$（ダブルプライム）をつける．e_{in} を input.xml に対して評価するとき，1〜4 行目までの for 構文の中間結果として得られるノード系列は $(3, 6', 3, 12', 9, 6'', 9, 12'')$ である．この結果は，文書順になっておらず，重複が発生している．この中間結果に対して 5 行目のスラッシュ '/' による DDO 処理によって，ノードの並び替えと重複削除が行われると，最終的な評価結果は $(3, 9, 6', 12', 6'', 12'')$ となる．■

例 1 のように，一般に中間結果はノードが文書順にならず重複が発生している．そのような中間結果に DDO 処理を行うと，処理負荷が増大するという懸念がある．そこで石原ら [2] は，パス式の後ろにのみスラッシュ '/' が現れるように XQuery 式を変換する手法を提案している．ここでのパス式とは，空系列 `()`，記号`$`で始まる変数，ファイル名 arg の XML 文書を呼び出す `doc(`arg`)`，XML 文書の木構造をたどることでノードを指定するロケーションステップ $ep/\alpha :: \tau$ のみから成る式を指す．パス式はスラッシュ '/' によってノードを探索する際，木構造をたどることで文書順に重複なくノードを探索することが難しくないため，処理負荷が比較的に大きくならない．この手法によって石原らは，変換後の XQuery 式が実行時間と記憶容量を節約できる場合があることを確認している．本研究では，石原らの手法をもとに let 構文と ec 構文を扱える最適化手法を提案する．ここで，本研究の提案手法を用いた let 構文と ec 構文を含む XQuery 式の変換例を例 2 で示す．

例 2　e_{in} を変換して得られる，e_{in} と等価でスラッシュ '/' がパス式の後ろにのみ現れる XQuery 式 e_{out} を図 3 に示す．e_{out} は，1〜7 行目の前半部分と 8〜15 行目の後半部分に分かれる．まず前半部分では，1 行目の for 構文によって input.xml の全ノードを文書順に`$x` に束縛する．e_{out} の 2 行目の if 構文は，e_{in} の 1〜4 行目の評価結果を抽出するための条件式にあたる．この条件式によって，`$x` に束縛されたノードが，e_{in} の 1〜4 行目の評価結果に合致するかを判

```
1      (for $a in doc("input.xml")/r/a
2       return let $v := document{element n {doc("input.xml")/r/a/c}}
3              return for $c in $v/n/c
4                     return if ($c/e) then ($a/b, $c/e) else ()
5      )/self::node()
```

図 2 入力 XQuery 式 e_{in}

```
1    (for $x in doc("input.xml")/descendant-or-self::node()
2     return if (for $a in doc("input.xml")/r/a
3                return let $v := document{element n {doc("input.xml")/r/a/c}}
4                       return if ($x/self::b/../self::a/../self::r/../self::document-node())
5                              then for $c in $v/n/c
6                                   return if ($c/e) then $x else ()
7                              else ()) then $x else(),
8     for $a in doc("input.xml")/r/a
9     return let $v := document{element n {doc("input.xml")/r/a/c}}
10           return for $y in $v/descendant-or-self::node()
11                  return if (if ($y/self::e/../self::c/../self::n/../self::document-node())
12                             then for $c in $y/self::e/..
13                                  return if ($c/self::c/../self::n/../self::document-node())
14                                         then if ($c/e) then $y else ()
15                                         else () else ()) then $y else())
```

図 3 出力 XQuery 式 e_{out}

定し，合致すれば$x を返し，合致しなければ空系列 () を返す．後半部分も同様に，ec 木 n の全ノードを文書順に$y に束縛し，$e_{\mathrm{in}}$ の 1〜4 行目の評価結果に合致すれば$y を返し，合致しなければ空系列 () を返す．このようにして，e_{out} では DDO 処理を回避している．その結果，e_{out} の評価結果は $(3, 9, 6', 12', 6'', 12'')$ となり，e_{in} と同じ評価結果が得られる． ■

3 諸定義

3.1 XML 文書

XML 文書は，ラベル付き順序木によって表現される．順序木は，1 つのドキュメントノード v_0 と 1 つ以上の要素ノードを持つ．ドキュメントノードは，順序木のルートノードであり，子ノードを 1 つだけ持つ．ラベルは XML 文書中のノードの要素名である．XML 文書中でノードが出現する順序を文書順序と呼ぶ．なお，本研究では属性ノードやテキストノードは扱わない．また，各問合せに与えられる XML 文書は 1 つだけであると仮定する．

3.2 XQuery

本研究では図 4 の文法で示した XQuery 式を対象とする．図 4 は [3] で定義されている XQuery 構文の部分クラスにあたる．本要旨では，本研究で新たに加える let 構文と ec 構文についてのみ説明する．`let` *lvar* `:=` *ec* `return` *ee* は，式 *ec* の評価結果である系列を変数 *lvar* に束縛した上で，式 *ee* を評価した結果の系列を返す．`document{element` *label* {*eloc*}} は，式 *eloc* の評価結果を子として持つ要素名 *label* のノードをドキュメントノードの子として設定した ec 木を生成する．

```
e    ::= ee
      |  for fvar in ei return e
      |  if eloc then e else ()
      |  let lvar := ec return ee
ee   ::= () | fvar | lvar | doc(arg)
      |  (ee, ... ,ee)
      |  ee/α :: τ
      |  for fvar in ee return ee
      |  if ee then ee else ()
ei   ::= eloc
      |  for fvar in ei return ei
      |  if eloc then ei else ()
ec   ::= document{element label {eloc}}
eloc ::= () | fvar | doc(arg) | eloc/α :: τ
α    ::= self | child | parent
      |  descendant | descendant-or-self
      |  ancestor | ancestor-or-self
      |  following-sibling | preceding-sibling
      |  following | preceding
τ    ::= label | * | node() | document-node()
```

図 4 対象とする入力 XQuery 式クラス

4 石原らによる XQuery 問合せの最適化手法

ここでは，石原ら [2] の手法の変換の流れを示す．変換の流れは以下の 3 ステップから成る．

ステップ 1. Q_{ANDO} の生成
ステップ 2. 入力式を正規化
ステップ 3. 出力式の生成

以下では，それぞれのステップについて説明する．

ステップ1では，入力文書のすべてのノードを重複なく文書順に出力する XQuery 式 Q_{ANDO} を生成する．ここでは，以下の式を Q_{ANDO} として使用する．

```
for $x in doc(arg)/descendant-or-self::node()
return $x
```

ステップ2では，5つの工程で入力式を正規化する．

- ステップ 2-1. スラッシュ '/' を式の最内に移動
- ステップ 2-2. 系列式の分解
- ステップ 2-3. 出力変数 `$x` の導入
- ステップ 2-4. in に続く式の簡単化
- ステップ 2-5. 出力変数 `$x` を束縛する `for` を最外に移動

ステップ3では，入力文書のノードが入力式の評価結果に含まれる条件を正規化した式から抽出し，Q_{ANDO} に挿入する．

5 let 構文と ec 構文を含む XQuery 式の変換

ここでは，本研究で提案する let 構文と ec 構文を含む XQuery 式の変換手法を示す．本手法は，石原ら [2] の手法の流れをもとに新たに変換手順や変換規則を加えたものである．前述したように，本手法は図 4 で示す入力 XQuery 式クラスを対象とする．本研究では，XQuery 式内で ec 木を扱いやすくするために，let 構文を用いて生成した ec 木のドキュメントノードを変数 *lvar* に束縛する．また，ec 構文が複数現れると式が複雑になり，本手法の適用が困難だったため，本研究では入力 XQuery 式に対して「ec 木のドキュメントノードを変数 *lvar* に束縛する let 構文は式内で1つしか現れない」という制限を設ける．以下では，それぞれのステップについて説明する．

本手法のステップ1では，石原らの手法で用いられる入力文書に対する Q_{ANDO} に加えて，変数 *lvar* に束縛された ec 木のすべてのノードを重複なく文書順に出力する XQuery 式として以下の式を用意する．

```
for $y in lvar/descendant-or-self::node()
return $y
```

本手法のステップ2では，let 構文の外側に for 構文のみが現れるように変換する．for 構文の内側に ec 構文が現れるとき，for 構文の繰り返しによって ec 木が複数生成される場合がある．このとき，生成された複数の ec 木が全く同じ構成であっても，それらの ec 木はそれぞれ異なる木として扱われるため，for 構文と let 構文の位置を入れ替えることはできない．したがって，for 構文と let 構文の入れ替えは行わずに，let 構文の外側の式をできるだけシンプルになるように入力式を正規化する．ここで，let 構文の内側の式には石原らの手法が適用できるため，let 構文の外側の式に対する変換手順が新たに必要となる．そこで本手法では，入力式を正規化するステップ2を以下の8つの工程で構成する．なお，下線部分が本手法で新たに追加した変換手順であり，let 構文の外側の式を操作する変換手順にあたるのはステップ 2-1 とステップ 2-2 である．

- <u>ステップ 2-1.</u> let 構文の外側の `in` に続く式を簡単化
- <u>ステップ 2-2.</u> if 構文を let 構文の内側に移動
- ステップ 2-3. スラッシュ '/' を式の最内に移動
- ステップ 2-4. 系列式の分解
- ステップ 2-5. 出力変数 `$x` の導入
- ステップ 2-6. `in` に続く式の簡単化
- ステップ 2-7. 出力変数 `$x` を束縛する `for` を外側に移動
- <u>ステップ 2-8.</u> ec 木のノードを指定する出力変数を`$y` に変更

$$\frac{\texttt{if } eloc_1 \texttt{ then (let \$v := } ec \texttt{ return } ee\texttt{) else ()}}{\texttt{let \$v := } ec \texttt{ return (if } eloc_1 \texttt{ then } ee \texttt{ else ())}} \quad (\textsc{Swap If-Let})$$

$$\frac{\texttt{for \$x in } ep \texttt{ return } ee \qquad ep \text{ contains } lvar}{\texttt{for \$y in } ep \texttt{ return } ee[\texttt{\$x} \leftarrow \texttt{\$y}]} \quad (\textsc{Change Output Var})$$

図5 本手法で追加した変換規則

$$
\begin{aligned}
e &::= el \mid es \\
el &::= \texttt{for } fvar \texttt{ in } eloc \texttt{ return } el \\
&\quad\mid \texttt{let } lvar \texttt{ := } ec \texttt{ return } es \\
es &::= \texttt{(}ef\texttt{,} \ldots \texttt{,}ef\texttt{)} \\
ef &::= \texttt{for \$x in } ep \texttt{ return } ee \\
&\quad\mid \texttt{for \$y in } ep \texttt{ return } ee \\
e &::= ep \\
&\quad\mid \texttt{for } fvar \texttt{ in } ep \texttt{ return } ee \\
&\quad\mid \texttt{if } ee \texttt{ then } ee \texttt{ else ()} \\
ec &::= \texttt{document\{element } label \ \{eloc\}\} \\
eloc &::= \texttt{()} \mid fvar \mid \texttt{doc(}arg\texttt{)} \mid eloc/\alpha :: \tau \\
ep &::= \texttt{()} \mid fvar \mid lvar \mid \texttt{doc(}arg\texttt{)} \mid ep/\alpha :: \tau
\end{aligned}
$$

図6 ステップ2の出力構文

ここで，本手法で新たに追加した変換規則を図5に示す．Swap If-Let は，ステップ 2-2 で if 構文と let 構文を入れ替えるときに適用する．Change Output Var は，ステップ 2-8 で ec 木のノードを指定する出力変数を`$y` に変更するときに適用する．ステップ2の出力構文を図6に示す．

最後にステップ3では，ステップ2で正規化した式を変形し Q_{ANDO} を挿入する．まず入力式に let 構文が現れない場合は，石原らの手法と同様の手順で出力式を生成するため本要旨では説明を省略する．入力式に let 構文が現れる場合，このステップの入力式は，

$$
\begin{aligned}
e = {} & \texttt{(for } fvar \texttt{ in } eloc \texttt{ return)}^* \\
& \quad \texttt{let } lvar := ec \texttt{ return (}ef_1\texttt{,} \ldots \texttt{,}ef_n\texttt{)}
\end{aligned}
$$

の形であり，$outvar ::= \texttt{\$x} \mid \texttt{\$y}$ とすると，

$$
\begin{aligned}
ef_1 &= \texttt{for } outvar_1 \texttt{ in } ep_1 \texttt{ return } ee_1 \\
&\ \ \vdots \\
ef_n &= \texttt{for } outvar_n \texttt{ in } ep_n \texttt{ return } ee_n
\end{aligned}
$$

となる．ステップ 3 の最終的な出力式は，入力木のノードを返す式と ec 木のノードを返す式を並列に並べた系列式である．まず，入力木のノードを返す式は以下のとおりである．以下の式では，入力木のノードを呼び出す Q_{ANDO} が最外に置かれている．ここで $inv(ep, outvar)$ は，$outvar$ に関する ep の逆関数である [2]．

```
for $x in doc()/descendant-or-self::node()
return if ((for fvar in eloc return)* let lvar := ec
return (if inv(ep_1, outvar_1) then ee_1 else (),
          ...,
          if inv(ep_n, outvar_n) then ee_n else ()))
         then $x else ()
```

また，ec 木のノードを返す式は以下のとおりである．以下の式では，ec 木のノードを呼び出す Q_{ANDO} が let 構文に直後に置かれている．

```
(for fvar in eloc return)* let lvar := ec
return for $y in lvar/descendant-or-self::node()
return if (if inv(ep_1, outvar_1) then ee_1 else (),
            ...,
            if inv(ep_n, outvar_n) then ee_n else ())
         then $y else ()
```

さらに，上の入力木のノードを返す式の 3〜5 行目のうち，$y を含む if 構文を空系列に置き換える．また，上の ec 木のノードを返す式の 3〜5 行目のうち，$x を含む if 構文を空系列に置き換える．結果として，以下の式が最終的な出力式となる．

```
(for $x in doc()/descendant-or-self::node()
 return if ((for fvar in eloc return)* let lvar := ec
 return (if inv(ep, $x) then ee else (),
          ...,
          if inv(ep, $x) then ee else ()))
         then $x else (),
 (for fvar in eloc return)* let lvar := ec
  return for $y in lvar/descendant-or-self::node()
  return if (if inv(ep, $y) then ee else (),
            ...,
            if inv(ep, $y) then ee else ())
         then $y else ())
```

6 実験

ここでは，図 2 に示す XQuery 式 e_{in} と，e_{in} を本手法に従って変換して得られた図 3 に示す XQuery 式 e_{out} に対して，実行時間と使用メモリ量の比較を行う．使用した PC は，dynabook RX73/JBE である．実験環境は Intel Core i5-7200U @ 2.50GHz, RAM 16GB, Windows 10 であり，XML データベースシステムである BaseX 9.7.2[1] で計測する．パラメータ max_c を 1000, 2000,. . . , 10000 と設定し，r ノードが 2 個の a ノードを持ち，各 a ノードが 1 個の b ノードと max_c 個の c ノードを持ち，各 c ノードは d, e, f ノードを 0 個または 1 個持つ，XML ファイルを 10 個生成した．生成された 10 個の XML ファイルそれぞれに対して，e_{in} と e_{out} を 10 回ずつ実行した．表 1 に，e_{in} と e_{out} をそれぞれ 10 回実行した際の平均時間と平均使用メモリ量を示す．$max_c = 10000$ のときの e_{in} と e_{out} の実行時間を比較すると，e_{out} の実行時間が e_{in} の実行時間の 2 分の 1 以下になっており，変換後の式の方が実行時間が節約されたことがわかる．しかし，使用メモリ量は変換後の式の方が大きくなることもわかった．これは，変換後の式の方が ec 木を生成する回数が増えることが原因であると考えられる．また，石原ら [2] の実験と同様に，いくつかの入力式では，変換後の式の実行時間が変換前の式の実行時間よりも長くなる場合があることがわかった．使用メモリ量は，変換前と変換後で変化がない場合もあった．

表 1 e_{in} と e_{out} の実行結果

max_c	e_{in} の実行時間 (ms)	e_{out} の実行時間 (ms)	e_{in} の使用メモリ量 (MB)	e_{out} の使用メモリ量 (MB)
1000	320.2	386.8	8.7	18.6
2000	522.0	572.3	18.8	37.2
3000	773.1	714.8	27.4	54.6
4000	1095.3	892.7	36.2	72.8
5000	1519.1	1144.1	45.7	91.0
6000	1983.5	1270.6	54.6	110.2
7000	2523.7	1463.7	64.6	128.4
8000	3148.9	1610.1	71.8	146.0
9000	3892.8	1848.1	81.1	163.6
10000	4604.4	1989.3	89.6	182.8

7 まとめと今後の課題

本研究では，石原らが提案する，XQuery 式を DDO 処理を含まない等価な式に変換する手法で，let 構文と ec 構文が扱えるように新たな変換手順や変換規則を加えることで，対象クラスの拡張を行った．実験によって本手法で変換した XQuery 式が，変換前の式より実行時間を節約できる場合があることを確認した．ただし，変換後の式の方が使用メモリ量が大きくなるという結果から，本手法は十分な記憶容量が確保できる場合に有効であると考えられる．今後の課題として，本手法の実用性を確認するために，実用で使われているスキーマに沿った XML 文書に対して評価する実験も行う必要がある．

参考文献

[1] BaseX | The XML Database. https://basex.org/.

[2] Y. Ishihara, H. Kato, and T. Grust. XQuery optimization by avoiding node sorting and duplicate elimination. In *IEICE Technical Report, SS2015-83*, pp. 43–48, 2016.

[3] World Wide Web Consortium. W3C XML Query (XQuery). https://www.w3.org/XML/Query/.