Reverse XSLT

Seminar XSLT
ZHAW, Zürich

Florian Lüthi*

2. Juli 2014

*luethifl@students.zhaw.ch

Die Arbeit beschäftigt sich mit der Umkehrbarkeit von XSL-Transformationen, welche als Instanzen von Konvertern angesehen werden. Da Konvertierungen im Allgemeinen Informationsverlust zur Folge haben (wobei Informationsanreicherung mit Informationsverlust gleichzusetzen ist), sind solche Konvertierungen (und ergo XSL-Transformationen) im Allgemeinen nicht bijektiv. Es ist sogar so, dass im Normalfall jedes Element der Wertemenge auf unendlich viele Elemente der Definitionsmenge abgebildet werden kann.

In Spezialfällen ist es möglich, nicht-bijektive Teil-Transformationen, welche auf denselben Daten operieren, so im Definitionsbereich einzuschränken, dass die Umkehrung eine endliche Anzahl an Resultaten produziert. Das Hauptinstrument hierzu ist Prädikatenlogik und brute force. Ebenfalls hilfreich ist der Zuzug des formalen Schemas der Input-Daten, um beispielsweise Datentypen einschränken zu können.
Die Frage nach der Existenz der Umkehrung für jede XSL-Transformation kann bejaht werden, da XSLTuring-vollständig ist und ein Algorithmus zur Invertierung von Turingmaschinen existiert. Eine solche Umkehrung führt aber im Allgmeinen zu unendlich vielen Resultaten.

Das grundsätzliche Vorgehen zur Umkehrung von XSL-Transformationen besteht darin, aus Template-Definitionen XPathes zu konstruieren und umgekehrt. Da die Abhängigkeiten der Template-Definitionen untereinander nur implizit gegeben sind, führt eine vollständige Konstruktion zu einer exponentiell wachsenden Anzahl resultierender Templates und dadurch zu brute force während der Ausführung der so konstruierten Umkehrung. Teilweise Abhilfe schafft wiederum das Hinzuziehen des Schemas der Input-Daten.

Der Fokus für eine Weiterführung dieser Arbeit könnte sich dahingehend verschieben, dass nicht die Umkehrung selber im Vordergrund steht, sondern eher die Entscheidung, ob eine Konvertierung bijektiv (und daher im Grunde unnötig) ist oder nicht.

Inhaltsverzeichnis

1 Einführung

Diese Seminararbeit ist Teil des Seminars *XSLT* an der ZHAW Zürich im Frühlingssemester 2014. Die Arbeit beschäftigt sich mit Möglichkeiten, Umkehrungen von `XSL`-Transformationen zu finden, sowie mit der Antwort auf die Frage, warum dies im Allgemeinen nicht möglich und im schon im Speziellen äusserst mühsam ist.

Diese Arbeit erhebt bezüglich der Möglichkeiten sowie den Antworten auf die obige Frage in keinster Weise Anspruch auf Vollständigkeit, da dies ihren Rahmen bei Weitem sprengen würde.

Zur Illustration dient das Programm `Tronsfarmer`, welches im Verlauf dieses Dokuments beschrieben wird. Diese Arbeit geht davon aus, dass die grundlegenden Funktionsweisen von `XSLT` sowie von `XPath` bekannt sind.

Des Autors Dank gilt Daniel Liebhart für die Durchführung des Seminars.

2 Grundlagen

2.1 Umkehrbarkeit von Konvertierungen

`XSL`-Transformationen sind klassische Instanzen von Konvertern. Ein Konverter führt einen Umwandlungsprozess durch, welcher sowohl zu Informationsverlust als auch zu Informationsanreicherung führen kann [1]. Dadurch ergibt sich unmittelbar das erste Problem, dass verlustig gegangene Information durch die Umkehrung nicht wiederhergestellt werden kann. Ein solcher Informationsverlust kann entweder inhaltlicher Natur (beispielsweise durch Filtering der Ausgangsdaten) oder struktureller Natur sein (beispielsweise durch Weglassen der Ebenen in baumartigen Daten).

2.2 Informationsanreicherung = Informationsverlust

Obschon der intuitive Konsens herrscht, dass Informationsanreicherung im Allgemeinen etwas positives und Informationsverlust um Allgemeinen etwas negatives darstellt, sind beide Phänomene bezogen auf die Umkehrbarkeit von Konvertierungen ein- und dasselbe Problem; Informationsanreicherung bedeutet in der Tat nichts anderes, als die Information, dass andere Information nicht vorhanden ist, durch ebendiese Information (welche nur aus einem externen Kontext kommen kann) zu verbergen. Daraus folgt, dass bei einem Akt von Informationsanreicherung immer auch andere Information verloren geht. Diese Beobachtung erlaubt es uns, im weiteren Verlauf nur noch auf das Problem des Informationsverlusts einzugehen.

2.3 Bijektivität von Konvertierungen

Die Mengentheorie bezeichnet eine Abbildung als bijektiv (also umkehrbar), wenn eine vollständige Paarbildung zwischen Definitionsmenge und Zielmenge stattfindet [2]. Eine solche Abbildung wird Bijektion genannt. Auf Konverter bezogen bedeutet ergo eine bijektive Konvertierung, dass die Eindeutigkeit eines Datenfragments derjenigen seines konvertierten Pendants entspricht – sowohl in struktureller als auch inhaltlicher Hinsicht.

Abbildung 2.1: Unabhängig teilbares Datenfragment

```
<header constant="constant" />
<payload>
        <data>abcde</data>
</payload>
```

2.4 Bijektivität eines Tupels von Konvertierungen

Es kann folgendes beobachtet werden: Seien $f : A \to B$ und $g : C \to D$ Bijektionen. Dann ist $h : A \times C \to B \times D$ mit

$$h(a, c) = (f(a), g(c))$$

ebenfalls eine Bijektion. Umgemünzt auf die Bijektivität von Konvertierungen bedeutet das, dass eine Konvertierung bijektiv ist, wenn die Konvertierungen für sämtliche unabhängige Teilbereiche eines Datenfragments bijektiv sind.

Ausserdem wird uns dadurch ermöglicht, die Datenfragmente derartig zu unterteilen, dass die Mächtigkeit der Fragmentmengen handhabbar wird. Zur Illustration sei in der Abbildungen 2.1 ein Datenfragment gegeben, welches die folgenden Teil-Mächtigkeiten aufweist:

- Das `<data/>`-Element hat eine Mächtigkeit von $|\Sigma^*| = \infty$ (wobei Σ das Alphabet der erlaubten Zeichen für Text innerhalb von XML-Elementen bezeichnet);
- Das `<payload/>`-Element hat dieselbe Mächtigkeit wie das `<data />`-Element;
- Das `<header/>`-Element hat eine Mächtigkeit von 1.

Mächtigkeiten von 1 sind für uns hilfreich, denn sie erlauben uns, konstante strukturelle Information, welche während der ursprünglichen Konvertierung verloren gegangen ist, durch zusätzliches externes Wissen (im Fall von `XSLT` beispielsweise ein XML-Schema) wiederherzustellen.

2.5 Abhängige Teilbereiche von Datenfragmenten

Im Allgemeinen ist eine Datenkonvertierung nicht bijektiv. Aus einem konvertierten Datenfragment können also im Allgemeinen unendlich viele nicht-konvertierte Datenfragmente wiederhergestellt werden.

Durch geschicktes Kombinieren der Umkehrungen von Konvertierungen, die auf denselben Datenelementen operieren, ist es aber in Spezialfällen durch Prädikatenlogik möglich, die Anzahl der möglichen ursprünglichen Fragmente auf eine endliche Zahl zu reduzieren. Als Beispiel wollen wir ein Datenfragment (a, b) betrachten. Ausserdem wissen wir aus externer konstanter Quelle, dass für den Datentyp $(a, b) \in \mathbb{N}_+^2$ gilt. Wir

Tabelle 2.1: Prädikate

(a, b)	$a + b$	$a + b = x$	$b \mod a$	$b \mod a = y$	$a + b = x \wedge b \mod a = y$
$(1, 1)$	2	falsch	0	falsch	falsch
$(1, 2)$	3	falsch	0	falsch	falsch
$(1, 3)$	4	falsch	0	falsch	falsch
$\vdots$					
$(1, 9)$	10	wahr	0	falsch	falsch
$\vdots$					
$(2, 8)$	10	wahr	0	falsch	falsch
$(3, 7)$	10	wahr	1	wahr	wahr
$(4, 6)$	10	wahr	2	falsch	falsch
$(5, 5)$	10	wahr	0	falsch	falsch
$\vdots$					
$(6, 4)$	10	wahr	4	falsch	falsch
$(7, 3)$	10	wahr	3	falsch	falsch
$(8, 2)$	10	wahr	2	falsch	falsch
$(9, 1)$	10	wahr	1	wahr	wahr

kennen des weiteren dessen konvertiertes Pendant $(x, y) = (10, 1)$ sowie die Konvertierungen:

$$\begin{aligned} x &= a + b, \\ y &= b \mod a. \end{aligned}$$

Es liegt auf der Hand, dass beide Konvertierungen keine direkte Umkehrung haben. Wenn wir aber die möglichen Werte, die (a, b) gemäss der Typdefinition einnehmen kann, aufzählen und die Konvertierungen als Prädikate verwenden, kommen wir zum Ergebnis wie in Tabelle 2.1 und haben dadurch die Anzahl möglicher Ausgangsdatenfragmente von unendlich auf zwei reduziert. Dies ist natürlich eine theoretische Betrachtung – unter der Annahme, dass die Menge $\mathbb{N}_+$ dem Datentyp `int` mit Einschränkung auf die positiven Zahlen entspricht, gäbe es nur schon für das erste Prädikat $(2^{31} - 1)^2$ Möglichkeiten zu überprüfen.

3 Konstruktion von Umkehrfunktionen von XSL-Teiltransformationen

In den nachfolgenden Abschnitten wollen wir die Konstruktion von einigen spezifischen Umkehrfunktionen für XSL-Teiltransformationen genauer unter die Lupe nehmen und anhand der Implementation im Programm Tronsfarmer erklären. Als Beispiel-Datenfragment dient uns jeweils das CD Catalog-XML von W3C (siehe Abbildung 3.1) sowie ein passendes Stylesheet (Abbildung 3.2).

3.1 <value-of/> innerhalb von <template/>

<value-of/> extrahiert den Wert eines Elements und fügt ihn an dieser Stelle in die Ausgabe ein [7].

Wir wollen uns auf die Extraktion des Wertes des aktuellen Elements (.) beschränken und erreichen eine Umkehrung, indem wir die Elementhierarchie bis zum umgebenden <template/>-Element hochtraversieren und in einen XPath umwandeln. Ausserdem werden alle Attribute sowie alle Nachbarelemente dieser Elementhierarchie als Prädikate formuliert. Das folgende Beispiel illustriert das Vorgehen, wobei aus dem ursprünglichen <template/>-Inhalt:

```
Artist: <span style="color:#00ff00">
  <xsl:value-of select="."/>
</span>
<br />
```

der folgende XPath wird:

```
//text()[
    parent::node()[
        name() = 'span'
        and @style='color:#00ff00'
        and preceding-sibling::text()[
            contains(., 'Artist:')
        ]
        and following-sibling::br
    ]
]
```

Dieser XPath wird dann zum match-Attribut des resultierenden Templates.

In Listing 3.3 ist die Methode zur Traversierung des Elementpfads gegeben.

Abbildung 3.1: Auszug aus CD Catalog

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
        <cd>
                <title>Empire Burlesque</title>
                <artist>Bob Dylan</artist>
                <country>USA</country>
                <company>Columbia</company>
                <price>10.90</price>
                <year>1985</year>
        </cd>
        <cd>
                <title>Hide your heart</title>
                <artist>Bonnie Tyler</artist>
                <country>UK</country>
                <company>CBS Records</company>
                <price>9.90</price>
                <year>1988</year>
        </cd>
        <cd>
                <title>Greatest Hits</title>
                <artist>Dolly Parton</artist>
                <country>USA</country>
                <company>RCA</company>
                <price>9.90</price>
                <year>1982</year>
        </cd>
        <!-- (...) -->
</catalog>
```

Abbildung 3.2: XSL für CD Catalog

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <body>
        <h2>My CD Collection</h2>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="cd">
    <p>
      <xsl:apply-templates select="title"/>
      <xsl:apply-templates select="artist"/>
    </p>
  </xsl:template>

  <xsl:template match="title">
    Title: <span style="color:#ff0000">
      <xsl:value-of select="."/>
    </span>
    <br />
  </xsl:template>

  <xsl:template match="artist">
    Artist: <span style="color:#00ff00">
      <xsl:value-of select="."/>
    </span>
    <br />
  </xsl:template>

</xsl:stylesheet>
```

Abbildung 3.3: `GetUpstairsXPath()`

```
1  private static string GetUpstairsXPath(XmlElement elem, XmlElement root,
       XmlNamespaceManager nsmgr) {
2    if (elem == root) {
3      return "";
4    }
5    if (elem.NamespaceURI == "http://www.w3.org/1999/XSL/Transform") {
6      return GetUpstairsXPath((XmlElement) elem.ParentNode, root, nsmgr);
7    }
8
9    var xpath = "parent::node()[name() = '" + elem.LocalName + "']";
10
11   // attributes
12   xpath += elem.Attributes.OfType<XmlAttribute>().Aggregate("", (current, attr)
       => current + (" and @" + attr.LocalName + "='" + attr.Value + "'"));
13
14   // preceding siblings
15   foreach (var x in elem.SelectNodes("./preceding-sibling::*").OfType<XmlNode>()
         .Union(elem.SelectNodes("./preceding-sibling::text()")
         .OfType<XmlNode>()).Reverse()) {
16     xpath += " and preceding-sibling::" + (x.NodeType == XmlNodeType.Text ?
         "text()" : x.LocalName);
17     if (x.NodeType == XmlNodeType.Text) {
18       xpath += "[contains(., '" + x.Value.Trim() + "')]";
19     } else if (x.Attributes.Count > 0) {
20       xpath += "[" + string.Join(" and ",
           x.Attributes.OfType<XmlAttribute>().Select(a => a.LocalName + "='" +
           a.Value + "'")) + "]";
21     }
22   }
23
24   // following siblings
25   foreach (var x in elem.SelectNodes("./following-sibling::*").OfType<XmlNode>()
         .Union(elem.SelectNodes("./following-sibling::text()") .OfType<XmlNode>()))
         {
26     xpath += " and following-sibling::" + (x.NodeType == XmlNodeType.Text ?
         "text()" : x.LocalName);
27     if (x.NodeType == XmlNodeType.Text) {
28       xpath += "[contains(., '" + x.Value.Trim() + "')]";
29     } else if (x.Attributes.Count > 0) {
30       xpath += "[" + string.Join(" and ",
           x.Attributes.OfType<XmlAttribute>().Select(a => a.LocalName + "='" +
           a.Value + "'")) + "]";
31     }
32   }
33
34   var up = GetUpstairsXPath((XmlElement) elem.ParentNode, root, nsmgr);
35   if (up != "") {
36     xpath += " and " + up;
37   }
38
39   xpath += "]";
40
41   return xpath;
42 }
```

3.2 match von <template/>

Dem match-XPath widerfährt das entgegengesetzte Schicksal – er wird zur resultierenden Elementhierarchie innerhalb derer das resultierende Template angewandt wird, wobei Attributprädikate zu Attributen und Elementprädikate zu Elementen, die nicht direkt auf dem Pfad liegen, werden. So wird beispielsweise aus dem ursprünglichen Template

```
<xsl:template match="//cd[title = 'Hide your heart']/artist">
  <!-- (...) -->
</xsl:template>
```

die folgende Elementhierarchie:

```
<cd>
  <title>Hide your heart</title>
  <artist>
    <!-- (...) -->
  </artist>
</cd>
```

3.3 Parsing von XPath

Das Parsen von XPath ist eine Wissenschaft für sich – wir verwenden dafür die Library XPathParser von Sergey Dubinets ([6]). Diese verwendet das Builder-Pattern ([6, 4]), wobei der Parser selber als Director fungiert und den XPath in einen Parsebaum überführt. Dieser Parsebaum ist eine DOM-Struktur und kann darum en globo analysiert werden, wie in Abbildung 3.4 illustriert wird.

Zwei Aspekte sind hierbei erwähnenswert:

1. Der nackte Root-XPath (/) ist eine nicht-bijektive Konvertierung, da er auf ein x-beliebiges Element passt. Um trotzdem überhaupt ein Resultat erzielen zu können, nennt GetElementFromPath() ein derartiges Element any-given-element.

2. Die Behandlung von Elementen in einer Achse ist verschieden, je nachdem ob sich diese Achse in einem Prädikat befindet oder nicht. Falls nicht, ist die Weiterführung des Templates (<apply-templates /> oder value-of) im letztmöglichen solchartiger Elemente anzusiedeln. Falls doch, darf genau diese Weiterführung dort aber nicht stattfinden.

3.4 Verknüpfung von <template>s

Um eine brauchbare Konstruktion zu erhalten, ist es erforderlich, eine Umkehrfunktion zu <apply-templates> (mit und ohne explizites select) zu entwickeln. Dies würde den Rahmen dieser Arbeit sprengen, würde aber so vonstatten gehen, dass für jede Permutation von Elementen der Potenzmenge aller XPathes entsprechende XPathes fomuliert werden. Dies ist notwendig, da die Verknüpfung der <template>s untereinander in der

Abbildung 3.4: `GetElementFromPath()`

```
1  private static Tuple<XmlElement, List<XmlElement>> GetElementFromPath(XElement
       path, XmlDocument owner) {
2    if (path.Name == "Child" && path.Attribute("nodeType").Value == "Element") {
3      var elem = owner.CreateElement(path.Attribute("name").Value);
4      var children = new List<XmlElement>();
5      if (!path.Elements().Any()) {
6        children.Add(elem);
7      }
8      foreach (var ch in path.Elements()) {
9        var che = GetElementFromPath(ch, owner);
10       if (che != null) {
11         elem.AppendChild(che.Item1);
12         children.AddRange(che.Item2);
13       }
14     }
15
16     return Tuple.Create(elem, children);
17   }
18
19   if (path.Name == "Root") {
20     var elem = owner.CreateElement("any-given-root");
21     var children = new List<XmlElement>();
22     if (!path.Elements().Any()) {
23       children.Add(elem);
24     }
25     foreach (var ch in path.Elements()) {
26       var che = GetElementFromPath(ch, owner);
27       if (che != null) {
28         elem.AppendChild(che.Item1);
29         children.AddRange(che.Item2);
30       }
31     }
32
33     return Tuple.Create(elem, children);
34   }
35
36   return null;
37 }
```

ursprünglichen `XSL`-Transformation nur implizit gegeben ist und erst durch die Anwendung auf das konkrete Datenfragment zu einer expliziten Verknüpfung wird. In der Konsequenz führt dies zu einer exponentiellen Anzahl Templates, was die Anwendung der Umkehrung auf die konvertierten Daten ebenfalls brute force werden lässt.

Diesem Problem könnten wir teilweise Abhilfe schaffen, indem wir das XML-Schema des ursprünglichen Datenfragments heranziehen würden.

4 Schlussfolgerungen und Ausblick

Es ist keine Überraschung, dass XSL-Transformationen im Allgemeinen nicht umkehrbar sind. Der prinzipielle formale Grund ist, dass Konvertierungen im Normallfall nicht bijektiv sind. Obschon Spezialfälle existieren, für welche es möglich ist, scheinbar nichtbijektive Konvertierungen entweder in explizit bijektive überzuführen oder zumindest deren Freiheitsgrade so einzuschränken, dass sie eine endliche Anzahl von Resultaten produzieren, führt die in dieser Arbeit skizzierte Vorgehensweise zwangsläufig zu brute force-Methoden, welche in der Praxis nur sehr eingeschränkt tauglich sind.

Die allgemeine, automatisierte Konstruktion von Umkehrungen gestaltet sich ebenfalls als schwierig. Die Gründe dafür bestehen vor allem in der Natur von XSLT, welche es erlaubt, die Verknüpfung von Templates nur implizit vorzunehmen. Dies führt dazu, dass die resultierende Umkehrung der Transformation selber zu einer brute force-Geschichte wird. Erschwerend (aber nicht verunmöglichend) kommt hinzu, dass XPath es erlaubt, quer zur Hierarchie zu arbeiten. Aus diesem Grund existiert auch keine Normalform für XSL-Transformationen und daher auch kein Algorithmus, welcher eine solche Transformation in ihre Normalform bringen würde (was die Entwicklung einer Umkehrung erleichtern würde).

Da XSL Turing-vollständig ist [5] und eine Konstruktionsmethode für inverse Turingmaschinen existiert [3], kann aber gesagt werden, dass es grundsätzlich möglich ist, Umkehrungen solcher Transformationen zu konstruieren. Diese werden aber nicht zu jedem konvertierten Datenfragment sein unkonvertiertes Pendant liefern, sondern alle möglichen solcher unkonvertierten Datenfragmente (in der Analogie der Turingmaschine finden sie für ein gegebenes Output-Tape alle möglichen Input-Tapes). Im Allgemeinen sind das jeweils unendlich viele, und damit können wir in den konkreten Problemstellungen der Datenverarbeitung (für welche XSLkonzipiert ist) normalerweise nichts sinnvolles anfangen.

Als Weiterführung dieser Arbeit könnte die Entwicklung eines Algorithmus oder einer Methodik in Betracht gezogen werden, welche allgemein entscheidet, ob eine gegebene XSL-Transformation bijektiv ist. Ein solcher Algorithmus könnte beispielsweise in unübersichtlichen, gewachsenen Systemlandschaften eingesetzt werden, um unnötige Transformationsschritte oder ganze unnötige Systeme zu identifizieren.

Literaturverzeichnis

[1] Daniel Liebhart. Information Engineering, Kapitel 6: Converter, 2014.

[2] H.D. Ebbinghaus. *Einführung in die Mengenlehre*. Spektrum Hochschultaschenbücher. Spektrum Akademischer Verlag, 2003.

[3] James Gips. A construction for the inverse of a turing machine. Technical report, Stanford, CA, USA, 1973.

[4] Object Oriented Design Forum. Builder Pattern. http://www.oodesign.com/builder-pattern.html.

[5] Ruhsan Onder and Zeki Bayram. Xslt version 2.0 is turing-complete: A purely transformation based proof. In *Proceedings of the 11th International Conference on Implementation and Application of Automata*, CIAA'06, pages 275–276, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Sergey Dubinets. XPath Parser. http://xpathparser.codeplex.com/.

[7] W3C. XSLT value-of Element. http://www.w3schools.com/xsl/xsl_value_of.asp.