On Modular Transformation of Structural Content

April 1, 2003

Tyng-Ruey Chuang
Institute of Information Science
Academia Sinica
Nangang 115, Taipei City, Taiwan
trc@iis.sinica.edu.tw

Jan-Li Lin
Institute of Information Science
Academia Sinica
Nangang 115, Taipei City, Taiwan
ljl@iis.sinica.edu.tw

ABSTRACT

We show that an XML DTD (Document Type Definition) can be viewed as the fixed point of a parametric content model. We then use natural transformations from the source content model to the target content model to derive DTD-aware and validated XML document transformations. Benefits of such transformations include static type-checking of XML transformational programs, automatic validation of target documents, and modular compositions of XML document transformers.

We prototype these modular XML document transformations in Objective Caml. The prototype depends heavily on the parametric module system of Objective Caml and is highly modular. Using Objective Caml to model XML document transformation also allows one to access high-level language constructs and supporting libraries in Objective Caml, hence enhance one's productivity in XML programming.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification Techniques.

General Terms

Design, Languages, Theory.

Keywords

Bird-Meertens formalism, document transformation and validation, functional programming, modules and interfaces, ML, Schema, XML.

1. BACKGROUND

Issues about XML document transformation have attracted much research attention in recent years. As XML documents are often quite complex, one needs certain guidelines in order to express, clearly, the transformation one has in mind when processing them. Furthermore, one often requires the transformation procedures to have certain nice properties, so that one can be sure of the quality of the output documents, as well as the efficiency of the transformation procedures themselves. For example, we may require the resulting documents to be valid with respect to a certain DTD, and we may demand the transformation procedure make no unnecessary traversal on the input or intermediate documents during the transformation.

There have been several proposals about XML query/processing languages [9, 11, 12]. Many of these languages provides XML-native syntax so one can program in concise expressions the intended transformation. There are also XML programming API in C++, Java, or other languages, to help people process XML documents using conventional programming languages [1]. What is less developed, however, is an XML transformational framework where one can express and prove properties of XML transformation functions. For example, one may want to know whether two succeeding XML document transformations can be fused into one so that it suffices to traverse the input document only once.

The work reported in this paper is related to the following work. XDuce is an typed XML processing language that produces valid XML documents [12, 13]. Milo, Suciu, and Vianu treated an XML transformer as a k-pebble transducer whose type-correctness can be checked against an output DTD [19]. Both of the above two languages are first-order hence cannot express composition of XML transformations in the languages themselves. Kuikka and Penttonen surveyed grammar-based, tree transformational methods for transforming structured documents [15]. Akpotsui, Quint, and Roisin worked on data type modeling for document transformation in a structured editing systems [5]. Neven and Van den Bussche discussed the expressiveness of attribute grammar based document transformation [22]. A data model for XML document transformation was discussed in Murata [20, 21], where an approach based on forest automata is proposed. Murata also advocated the idea of document transformation based on schema transformation.

Our work is inspired by Murata's. However, we do not use tree automata, nor attribute grammars. Rather, we formulate DTD-aware XML document transformations in a more abstract setting, often borrowing notations and results from category theory [16]. Our approach is algebraic [8], and can

This research is supported, in part, by National Science Council of Taiwan (contract no. NSC 89-2219-E-001-005, 90-2219-E-001-004, and 91-2219-E-001-006) and by the Institute of Applied Science and Engineering Research, Academia Sinica, Taiwan.
This paper appears as technical report TR-IIS-03-007 from the Institute of Information Science, Academia Sinica, Nangang 115, Taipei City, Taiwan. The electronic version of this paper is available from the Institute's website at <http://www.iis.sinica.edu.tw>, or from the authors by e-mail (trc@iis.sinica.edu.tw).

be viewed as extending the Bird-Meertens formalism [7, 17] to functions that map between two sets of mutually recursive data types. In this paper, we freely use the notations of Meijer, Fokkinga, and Paterson when writing fold/unfold functions [18].

We have prototyped the theory developed in this paper in Objective Caml [3], a functional language that supports both a polymorphic type scheme and a parametric module system. Our construction is highly modular and generic. It is generic because it is parameterized by DTD expressions. (See [6] for a detailed introduction to generic programming.) This work is a continuation of our previous work on generic validation of structural content [10], where we show how untyped (i.e., well-formed) XML expressions can be type-checked (i.e., validated) almost automatically in a higher-order functional language that supports both a polymorphic type scheme and a parametric module system. This paper shows how to model and compose typed XML transformational functions in such a functional language.

2. PARAMETRIC CONTENT MODELS

In an XML element type declaration, an element type is given a content model which is a regular expression of element type names. [1] Only element sequences that are derivable from the regular expression are allowed as the children of the element. For example, the following XML document contains a DTD that defines two element types `folder` and `record`. The document contains as a root a `folder` element, which has an empty `record` element as its only child. It is a valid XML document.

```
<?xml version="1.0"?>
<!DOCTYPE folder [
<!ELEMENT folder ((record,(folder|record)*)|
                  (folder,(folder|record)+))>
<!ELEMENT record EMPTY>
]>
<folder><record></record></folder>
```

The above DTD models the structure where a record must contain no other element, and no folder is ever empty or contains just another folder. (One may think of it as modeling a tidy bookmark file.)

Although an element content model is a regular expression of element type names, we may treat the element type names in the regular expressions as parameters. We call them *parametric content models*, and the declared element types are just the fixed points of the parametric content models. As an example, for the element type declarations in the `folder` DTD, we can define the following two parametric content models F_1 and F_2, of which each takes two element types as arguments:

$$
\begin{aligned}
F_1(x_0, x_1) &= (x_1, (x_0|x_1)*)|(x_0, (x_0|x_1)+) \\
F_2(x_0, x_1) &= \epsilon
\end{aligned}
$$

The element types *folder* and *record* are the simultaneous fixed points of F_1 and F_2:

$$
\begin{aligned}
\mathit{folder} &= F_1(\mathit{folder}, \mathit{record}) \\
\mathit{record} &= F_2(\mathit{folder}, \mathit{record})
\end{aligned}
$$

Treating element types as fixed points of parametric content models allow us to use the parametric content models in the modeling and composition of XML document transformations. For document transformations that are based on mappings between different content models, we call them *DTD-aware* XML document transformations as they will necessarily involve the DTD of the source document and the DTD of the target document.

3. XML DOCUMENT TRANSFORMATION AS FOLD/UNFOLD

For simplicity, we use $s = (s_1, s_2, \ldots, s_n)$ to denote a DTD s consisting of a tuple of element types $s_1, s_2, \ldots, s_n$. The tuple $(s_1, s_2, \ldots, s_n)$ is understood as the simultaneous fixed point of a tuple of parametric content models $P = (P_1, P_2, \ldots, P_n)$. That is,

$$
\begin{aligned}
(s_1, s_2, \ldots, s_n) &= (P_1(s_1, s_2, \ldots, s_n), \\
&\quad P_2(s_1, s_2, \ldots, s_n), \\
&\quad \ldots, \\
&\quad P_n(s_1, s_2, \ldots, s_n))
\end{aligned}
$$

We use $s = Ps$ to denote s as the fixed point of P. Let $\mathrm{up}_s : Ps \rightarrow s$ and $\mathrm{down}_s : s \rightarrow Ps$ be the two mappings that together define the identities

$$
\begin{aligned}
\mathrm{up}_s \circ \mathrm{down}_s &= \mathrm{id}_s \\
\mathrm{down}_s \circ \mathrm{up}_s &= \mathrm{id}_{Ps}
\end{aligned}
$$

Because P is parametric, one can define a function $Pf : Ps \rightarrow Pt$ whenever given a tuple of functions $f = (f_1, f_2, \ldots, f_n)$ with $f_i : s_i \rightarrow t_i$ for each i. Pf is understood as

$$
Pf = (P_1(f_1, f_2, \ldots, f_n), P_2(f_1, f_2, \ldots, f_n), \ldots, P_n(f_1, f_2, \ldots, f_n))
$$

where, for each i, $P_i(f_1, f_2, \ldots, f_n)$ is the function that maps $P_i(v_1, v_2, \ldots, v_n)$ to $P_i(f_1(v_1), f_2(v_2), \ldots, f_n(v_n))$. Moreover, by the parametricity of P, one can show that

$$
\begin{aligned}
P\, id_s &= id_{Ps}, \\
(Pg) \circ (Pf) &= P(g \circ f)
\end{aligned}
$$

for all $f : s \rightarrow t$ and $g : t \rightarrow u$. That is, P is a functor: It maps types s and t to types Ps and Pt, and maps typed function $f : s \rightarrow t$ to function $Pf : Ps \rightarrow Pt$.

Let $s = Ps$ and $t = Qt$ be two DTDs with the same arity, i.e., they each define exactly n element types. A function from s to t — i.e., an XML document transformation that maps documents of DTD s to documents of DTD t — is a

[1] An element can also has a *mixed content* model where its children is a sequence of XML elements interspersed with character data. As an example, `<!ELEMENT folder (#PCDATA|folder|record)*>` declares `folder` to be of mixed content. (#PCDATA denotes the interspersed character data.) One can still model `folder` by a regular expression (of two type variables) but now with an additional type constant #PCDATA.

fold function if it is characterized by a reduction function $f : Pt \rightarrow t$ with the following commutative diagram:

$$\begin{array}{ccc} s & \xleftarrow{\text{up}_s} & Ps \\ \Big\downarrow (\!|f|\!) & & \Big\downarrow P(\!|f|\!) \\ t & \xleftarrow[f]{} & Pt \end{array}$$

Similarly, the transformation is an unfold function if it is characterized by a generating function $g : s \rightarrow Qs$ with the following commutative diagram:

$$\begin{array}{ccc} Qs & \xleftarrow{g} & s \\ \Big\downarrow Q[\!(g)\!] & & \Big\downarrow [\!(g)\!] \\ Qt & \xleftarrow[\text{down}_t]{} & t \end{array}$$

That is,

$$\begin{array}{rcl} (\!|f|\!) \circ \text{up}_s & = & f \circ P(\!|f|\!) \\ \text{down}_t \circ [\!(g)\!] & = & Q[\!(g)\!] \circ g \end{array}$$

or, equivalently,

$$\begin{array}{rcl} (\!|f|\!) & = & f \circ P(\!|f|\!) \circ \text{down}_s \\ [\!(g)\!] & = & \text{up}_t \circ Q[\!(g)\!] \circ g \end{array}$$

4. NATURAL TRANSFORMATIONS BETWEEN CONTENT MODELS

We re-introduce the notion of *natural transformation*, a concept from category theory, and recast it in the context of XML document transformation. Natural transformations have been used to model polymorphic functions, see, e.g., [24]. Certain polymorphic functions, like the `map` function for the list data type, are also known to be expressible as both a fold function and as an unfold function. Here we make an explicit connection between the above two observations, showing that a natural transformation between two inductive bases P and Q is sufficient to define a function from $s = Ps$ to $t = Qt$ that is both a fold function and an unfold function.

Definition 4.1. Let P and Q denote two parametric content models of the same arity. A natural transformation η from P to Q is a collection of functions that is indexed by DTDs and satisfies the following equation

$$\eta_y \circ Pf = Qf \circ \eta_x$$

for any DTD x and y, and for any XML document transformation $f : x \rightarrow y$. That is, the following diagram commutes

$$\begin{array}{ccc} Px & \xrightarrow{\eta_x} & Qx \\ \Big\downarrow Pf & & \Big\downarrow Qf \\ Py & \xrightarrow[\eta_y]{} & Qy \end{array}$$

◇

One can think of the transformation $Pf : Px \rightarrow Py$ as the following: Apply f to the children x of the structural content Px in order to produce a structural content Py with children y. Likewise for transformation $Qf : Qx \rightarrow Qy$. A natural transformation $\eta : P \rightarrow Q$ changes the structure of the content tree but does not alter child elements. Function η is a natural transformation if it can be performed either after the transformation Pf or before the transformation Qf to arrive at same result. One may say η is polymorphic as it maps between parametric content models and does not look into the parameters, i.e., child elements, of the content.

Given two DTDs $s = Ps$ and $t = Qt$, and a natural transformation $\eta : P \rightarrow Q$, one can construct a fold function from s to t as $(\!|\text{up}_t \circ \eta_t|\!)$ as well as an unfold function from s to t as $[\!(\eta_s \circ \text{down}_s)\!]$. Furthermore, natural transformation η provides a sufficient condition for the fold function and the unfold function to coincide, as shown by the following Proposition.

Proposition 4.2. $(\!|\text{up}_t \circ \eta_t|\!) = [\!(\eta_s \circ \text{down}_s)\!]$. ◇

Proof.

$$\begin{array}{rcl} [\!(\eta_s \circ \text{down}_s)\!] & = & \text{up}_t \circ Q[\!(\eta_s \circ \text{down}_s)\!] \circ \eta_s \circ \text{down}_s \\ & & \quad \text{— definition of unfold} \\ & = & \text{up}_t \circ \eta_t \circ P[\!(\eta_s \circ \text{down}_s)\!] \circ \text{down}_s \\ & & \quad \text{— } \eta \text{ a natural transformation} \end{array}$$

By the uniqueness of fold, one also has

$$(\!|\text{up}_t \circ \eta_t|\!) = \text{up}_t \circ \eta_t \circ P(\!|\text{up}_t \circ \eta_t|\!) \circ \text{down}_s$$

It follows that $(\!|\text{up}_t \circ \eta_t|\!) = [\!(\eta_s \circ \text{down}_s)\!]$. ◇

Proposition 4.2 is best illustrated by the commutative diagram in Figure 1.

5. DEALING WITH MIS-MATCHED ARITIES

In the previous section, we have shown how to use a natural transformation from P to Q to derive a transformation from $s = Ps$ to $t = Qt$. A restriction is that P and Q must have the same arity — both models accept the same number of parameters. Otherwise there exists no natural transformation from P to Q. We are interested in the more general case where the arities of P and Q differ, so we can model more XML document transformations.

Let $P = (P_1, P_2, \ldots, P_m)$ be an m-ary content model, and $Q = (Q_1, Q_2, \ldots, Q_n)$ be an n-ary content model. Let $M =$

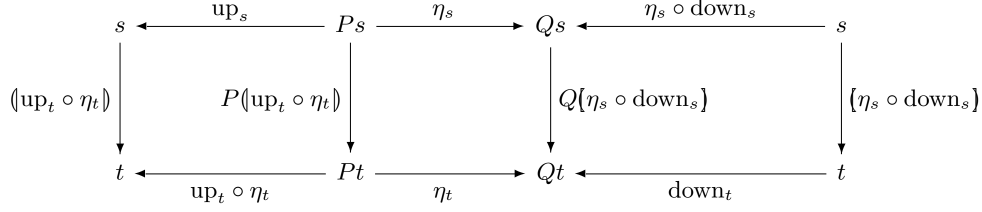


Figure 1: Commutative diagram for Proposition 4.2.

$\{1, 2, \ldots, m\}$ and $N = \{1, 2, \ldots, n\}$ be the two index sets. We now model an XML document transformation from $s = Ps$ to $t = Qt$ using an additional function $\sigma : M \to N$ between the two index sets. Function σ prescribes that elements of type s_i in the source document will be transformed into elements of type $t_{\sigma(i)}$ in the target document. In the previous section, one is restricted to the case where $M = N$ and σ is the identity function.

From σ one derives two functors F and G:

$$\begin{aligned} F(x_1, x_2, \ldots, x_m) &= (\oplus_{\sigma(j)=1} x_j, \ldots, \oplus_{\sigma(j)=i} x_j, \ldots, \oplus_{\sigma(j)=n} x_j) \\ G(y_1, y_2, \ldots, y_n) &= (y_{\sigma(1)}, \ldots, y_{\sigma(i)}, \ldots, y_{\sigma(m)}) \end{aligned}$$

where $\oplus_{\sigma(j)=i} x_j$ is the coproduct of those x_j's with $\sigma(j) = i$, that is, $j \in \sigma^{-1}(i)$. One immediate consequence is that for each function $h : x \to Gy$ there exists an equivalent function $k : Fx \to y$, and vice versa. F is called the *left adjoint* of G (equivalently, G is the *right adjoint* of F). We call functors like F and G *arity-adjusting functors* because F maps a m-tuple to a n-tuple and G maps a n-tuple to a m-tuple.

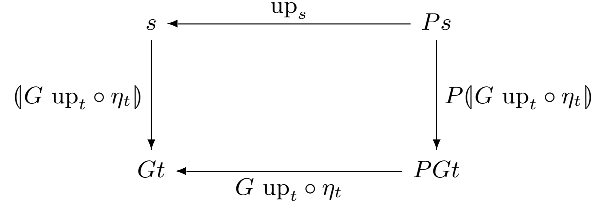
We list in Appendix A basic properties of adjoint functors, as as well proofs of additional propositions of arity-adjusting adjoint functors. For now, it suffices to know that from F and G, i.e., from σ alone, one can construct natural transformations $\varphi_{x,y}(h) = k$ and $\psi_{x,y}(k) = h$, where $h : x \to Gy$ is a m-tuple mapping and $k : Fx \to y$ a n-tuple mapping. Furthermore, the following identities hold:

$$\begin{aligned} \varphi_{x,y} \circ \psi_{x,y} &= \mathrm{id}_{Fx \to y} \\ \psi_{x,y} \circ \varphi_{x,y} &= \mathrm{id}_{x \to Gy} \end{aligned}$$

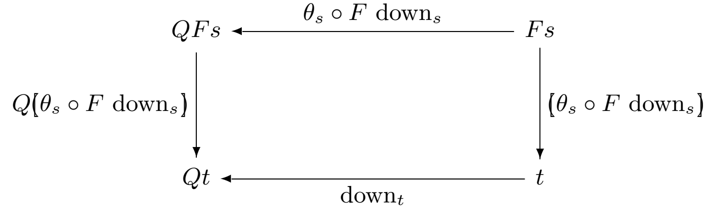
From φ and ψ one can derive two *universal arrows*. They are called *unit* and *counit*, and are natural transformations as well:

$$\begin{aligned} \mathrm{unit}_x : x \to GFx &= \psi_{x,Fx}(F\ \mathrm{id}) \\ \mathrm{counit}_y : FGy \to y &= \varphi_{Gy,y}(G\ \mathrm{id}) \end{aligned}$$

Recall that in the previous section, we have used a natural transformation $\eta : P \to Q$ to derive the transformation from $s = Ps$ to $t = Qt$ as a fold function and as an unfold function, but under the restriction that P and Q have the same arity. Now that P and Q have different arities, so instead we use a natural transformation $\eta : PG \to GQ$ to derive a transformation from s to Gt as a fold function:



Similarly, whenever given a natural transformation $\theta : FP \to QF$, one can derive a transformation from Fs to t as an unfold function:



We are ready to show a sufficient condition for the fold function and the unfold function to coincide.

PROPOSITION 5.1. $(\!|G\ \mathrm{up}_t \circ \eta_t|\!) = \psi_{s,t}[\!(\theta_s \circ F\ \mathrm{down}_s)\!]$ if $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$. ◇

PROOF. See Appendix A. ◇

PROPOSITION 5.2. $[\!(\theta_s \circ F\ \mathrm{down}_s)\!] = \varphi_{s,t}(\!|G\ \mathrm{up}_t \circ \eta_t|\!)$ if $\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy})$. ◇

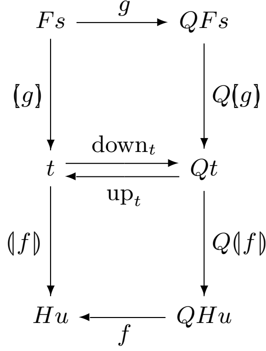
PROOF. See Appendix A. ◇

The above two propositions show that the fold function and the unfold function coincide if the two natural transformations coincide. Specifically, Proposition 5.1 shows that the fold function defined by a natural transformation $\eta : PG \to GQ$ is isomorphic to the unfold function defined by another natural transformation $\theta : FP \to QF$ if θ is derivable from η. Similarly, an unfold function defined by θ is a fold function defined by η if η is derivable from θ.

In summary, we model a transformation from $s = Ps$ to $t = Qt$ by a pair of adjoint functors F and G, as well as a natural transformation $\eta : PG \to GQ$ (equivalently, $\theta : FP \to QF$). The resulting s to t transformation can be viewed both as a fold function and as an unfold function.

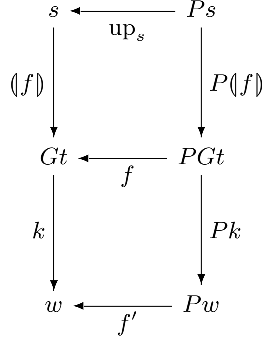
6. FUSING XML DOCUMENT TRANSFORMATIONS

In this section we show that the usual fusion laws in fold/unfold algebraic formalism can be translated to the setting of DTD-aware XML document transformations. Let $s = Ps$, $t = Qt$, and $u = Ru$ be DTDs, and let F, G, and H be arity-adjusting functors. The following diagram illustrates that an unfold function $\{\!|g|\!\}$ immediately followed by a fold function $(\!|f|\!)$ can be fused together. The fused function, $[\![f, g]\!] = (\!|f|\!) \circ \{\!|g|\!\}$, is known as *hylomorphism* [18].



FACT 6.1. $[\![f, g]\!] = f \circ Q[\![f, g]\!] \circ g$. (See [18].) $\diamond$

The following diagram illustrates the situation where a fold function can be fused with another function to become yet another fold function.



FACT 6.2. $k \circ (\!|f|\!) = (\!|f'|\!)$ if $k \circ f = f' \circ Pk$. (See [8].) $\diamond$

We now give a sufficient condition to fuse two XML document transformations into one.

PROPOSITION 6.3. Let $(\!|f|\!) : s \to Gt$ and $(\!|h|\!) : t \to Hu$ be two fold functions derived by, respectively, the two inductive functions $f : PGt \to Gt$ and $h : QHu \to Hu$. Then,

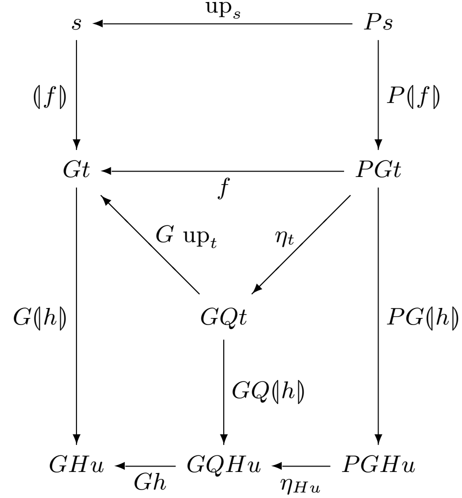
$$G(\!|h|\!) \circ (\!|f|\!) = (\!|Gh \circ \eta_{Hu}|\!)$$

if $f = G\ \text{up}_t \circ \eta_t$ for some natural transformation $\eta : PG \to GQ$. $\diamond$

PROOF. By Fact 6.2, it suffices to show the equation

$$G(\!|h|\!) \circ G\ \text{up}_t \circ \eta_t = Gh \circ \eta_{Hu} \circ PG(\!|h|\!)$$

holds in the following diagram.



But the equation follows directly from

$$\begin{aligned}
G(\!|h|\!) \circ G\ \text{up}_t &= Gh \circ GQ(\!|h|\!) \\
&\quad \text{— } (\!|h|\!) \text{ a fold function, } G \text{ a functor} \\
GQ(\!|h|\!) \circ \eta_t &= \eta_{Hu} \circ PG(\!|h|\!) \\
&\quad \text{— } \eta \text{ a natural transformation.}
\end{aligned}$$

$\diamond$

Notice that, in the above, function h is not required to be derivable from a natural transformation. It can be just any function from QHu Hu. However, if function h is characterized by a natural transformation $\zeta : QH \to HR$ such that $h = H\ \text{up}_u \circ \zeta_u$, then one has the additional benefit that

$$(\!|Gh \circ \eta_{Hu}|\!) = (\!|K\ \text{up}_u \circ \xi_u|\!),$$

where

$$\begin{aligned}
K &= GH \\
&\quad \text{— } K \text{ an arity-adjusting functor,} \\
\xi_x &= G\zeta_x \circ \eta_{Hx} \\
&\quad \text{— } \xi : PK \to KR \text{ a natural transformation.}
\end{aligned}$$

That is, the resulting XML document transformation, the fold function $(\!|Gh \circ \eta_{Hu}|\!)$, is again characterized by a natural transformation ξ. Furthermore, ξ can be automatically derived once ζ and η are given.

If one views the two XML document transformations as two unfold functions, one derives the dual case where the fused transformation is again an unfold function.

PROPOSITION 6.4. Let $\{\!|g|\!\} : Fs \to t$ and $\{\!|h|\!\} : Et \to u$ be two unfold functions derived by, respectively, the two generating functions $g : Fs \to QFs$ and $h : Et \to REt$. Then,

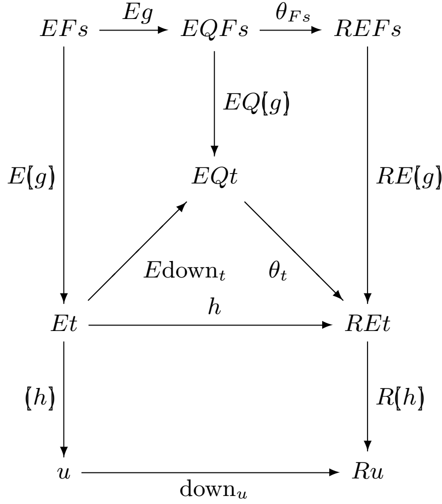
$$\{\!|h|\!\} \circ E\{\!|g|\!\} = \{\!|\theta_{Fs} \circ Eg|\!\}$$

if $h = \theta_t \circ E\ \text{down}_t$ for some natural transformation $\theta : EQ \to RE$. $\diamond$

PROOF. By the unfold fusion law, it suffices to show the equation

$$\theta_t \circ E\ \mathrm{down}_t \circ E\llparenthesis g \rrparenthesis = RE\llparenthesis g \rrparenthesis \circ \theta_{Fs} \circ Eg$$

holds in the following diagram.



But the equation follows directly from

$$\begin{array}{rcl} E\ \mathrm{down}_t \circ E\llparenthesis g \rrparenthesis & = & EQ\llparenthesis g \rrparenthesis \circ Eg \\ & & \quad \text{— } \llparenthesis g \rrparenthesis \text{ a unfold function, } E \text{ a functor} \\ \theta_t \circ EQ\llparenthesis g \rrparenthesis & = & RE\llparenthesis g \rrparenthesis \circ \theta_{Fs} \\ & & \quad \text{— } \theta \text{ a natural transformation.} \end{array}$$

◇

Again, the resulting unfold function is also characterized by a natural transformation if both g and h are characterized by natural transformations.

7. FURTHER GENERALIZATION WITH ATTRIBUTES

The element type definitions in an XML DTD may be accompanied with attribute-list declarations. For example, one may require a `record` element to be annotated with additional attributes, such as `title` and `url`, as shown below.

```
<?xml version="1.0"?>
<!DOCTYPE folder [
<!ELEMENT folder ((record,(folder|record)*)|
                  (folder,(folder|record)+))>
<!ELEMENT record EMPTY>
<!ATTLIST folder
          subject CDATA #IMPLIED>
<!ATTLIST record
          title   CDATA #REQUIRED
          url     CDATA #REQUIRED>
]>
<folder subject="Research Institutes">
   <record title="Academia Sinica"
           url="http://www.sinica.edu.tw"></record>
</folder>
```

As a consequence, a DTD $s = (s_1, s_2, \ldots, s_n)$ now is expressed as the simultaneous fixed point of the parametric content model $A \bullet P$, where

$$\begin{array}{rcl} (A \bullet P)(x_1, x_2, \ldots, x_n) & = & (A_1 \otimes P_1(x_1, x_2, \ldots, x_n), \\ & & \ A_2 \otimes P_2(x_1, x_2, \ldots, x_n), \\ & & \ \ldots, \\ & & \ A_n \otimes P_n(x_1, x_2, \ldots, x_n)) \end{array}$$

where A_i is for attributes of the ith element type, and $\otimes$ is the product operator.

The natural transformation $\eta : PG \to GQ$ that is used previously to characterize a function from s to Gt as a fold will now have type $\eta : A \bullet (PG) \to G(B \bullet Q)$. Furthermore, the index set mapping function $\sigma : M \to N$ that defines the adjoint functors F and G for mapping between P and Q will now be generated to have type: $\sigma : M \to 2^N$, where 2^N is the power set of N. This is because now an element of one type can be mapped to elements of more than one type, depending on the element's attribute values.

From the new σ one also derives two new functors F and G:

$$\begin{array}{rcl} F(x_1, x_2, \ldots, x_m) & = & (\oplus_{1\in\sigma(j)} x_j, \ldots, \oplus_{i\in\sigma(j)} x_j, \ldots, \oplus_{n\in\sigma(j)} x_j) \\ G(y_1, y_2, \ldots, y_n) & = & (\oplus_{j\in\sigma(1)} y_j, \ldots, \oplus_{j\in\sigma(i)} y_j, \ldots, \oplus_{j\in\sigma(m)} y_j) \end{array}$$

As before, the natural transformation $\eta : A \bullet (PG) \to G(B \bullet Q)$ will induce a fold function from s to Gt. Similarly, the natural transformation $\theta : F(A \bullet P) \to B \bullet (QF)$ will induce a unfold function from Fs to t. However, the modularity results in Section 6 (Propositions 6.3 and 6.4) also hold.

Note that functors F and G are not adjoint functors in the usual category (of which objects are fixed-arity products and arrows are element-wise total functions). However, if we allow arrows in the category to be element-wise *partial functions*, then in a limited sense, F and G are still adjoint to each other.

8. AN ILLUSTRATING EXAMPLE

Let us consider the following three DTDs:

```
<!ELEMENT even (odd?)>          -- DTD s
<!ELEMENT odd  (even)>

<!ELEMENT succ (succ?)>         -- DTD t

<!ELEMENT list_even (one*)>     -- DTD u
<!ELEMENT list_odd  (one*)>
<!ELEMENT one       EMPTY>
```

DTD s specifies that an `even` element contains either 0 or 1 `odd` element, and an `odd` element contains exactly 1 `even` element. DTD t specifies that a `succ` element contains either 0 or 1 `succ` element. DTD u specifies that a `list_even` element contains any number of `one` elements, a `list_odd` element contains any number of `one` elements, and a `one` element contains no element. Note that DTDs s, t, and u are defined, respectively, as the fixed points of the following parametric content models P, Q, and R (also shown with

their Objective Caml type definitions):

$$\begin{array}{rclcl} P_1(x_1, x_2) & = & x_2? & = & x_2 \texttt{ option} \\ P_2(x_1, x_2) & = & x_1 & = & x_1 \\ \\ Q_1(y_1) & = & y_1? & = & y_1 \texttt{ option} \\ \\ R_1(z_1, z_2, z_3) & = & z_3* & = & z_3 \texttt{ list} \\ R_2(z_1, z_2, z_3) & = & z_3* & = & z_3 \texttt{ list} \\ R_3(z_1, z_2, z_3) & = & \epsilon & = & \texttt{unit} \end{array}$$

Now consider the following two transformations:

T1 A pair of functions f_1 and f_2 that transform, respectively, an `even` element to a `succ` element and an `odd` element to a `succ` element. Both functions maintain the "sizes" of the input arguments, e.g. f_1 maps <even></even> to <succ></succ>, and f_2 maps <odd><even><odd><even></even></odd></even></odd> to <succ><succ><succ><succ></succ></succ></succ> </succ>.

T2 A function g that transforms a `succ` element to the corresponding sequence of `one` elements, which are enclosed within a `list_even` element if the total number of `one` elements is even. Otherwise the sequence of `one` elements is enclosed within a `list_odd` element. That is, <succ> </succ> is mapped to <list_even> </list_even>, and <succ><succ><succ><succ></succ></succ></succ> </succ> is mapped to <list_odd><one></one> <one></one><one></one></list_odd>.

Both transformations can be viewed as as fold operations. The rest of this section details the construction, as well as how the two folds can be fused. The first transformation has the index set mapping $\sigma_{s\to t}(1) = \{1\}, \sigma_{s\to t}(2) = \{1\}$, while the second transformation has the index set mapping $\sigma_{t\to u}(1) = \{1, 2\}$. These two index mappings specify two arity-adjusting functors G and H as follows.

$$\begin{array}{rcl} G(y_1) & = & (y_1, y_1) \\ H(z_1, z_2, z_3) & = & z_1 \oplus z_2 \end{array}$$

We need functor G to characterize the pairs (f_1, f_2) as a fold operation induced by a natural transformation $\eta : PG \to GQ$. Similarly, H is needed in order for g to be characterized as a fold induced by a function $\zeta : QH \to HR$.

Now that we have functors P, Q, R, G, and H, the types of η and ζ are worked out to be

$$\begin{array}{rcl} \eta_y & : & (P_1(G(y_1) \to Q_1(y_1)) \otimes (P_2(G(y_1) \to Q_1(y_1)) \\ \zeta_z & : & Q_1(H(z_1, z_2, z_3)) \to H(R_1(z_1, z_2, z_3), \\ & & \quad R_2(z_1, z_2, z_3), R_3(z_1, z_2, z_3)) \\ & & \text{with } z_1 = u_1, z_2 = u_2, \text{ and } z_3 = u_3 \end{array}$$

Note that ζ is not a natural transformation as we need additional constraints that $z_1 = u_1, z_2 = u_2$, and $z_3 = u_3$. This does not affect the fusion of **T1** and **T2** because, as shown in Proposition 6.3, ζ need not be a natural transformation.

What do the definitions of η and ζ look like? η is a pair of functions so let us call its two components by η_1 and η_2. η_1 assembles the content for a `succ` element from the content of an `even` element, assuming the latter already consists of, if any, a `succ` element. Likewise, η_2 assembles the content for a `succ` element from the content of an `odd` element, assuming the latter must already consist of a `succ` element. Again, ζ assembles the content for a `list_even` or a `list_odd` element from the content of a `succ` element, assuming the latter already consists of, if any, a `list_even` element or a `list_odd` element. Because both transformations **T1** and **T2** maintain the sizes of the input arguments, we can arrive at the Objective Caml code in Figure 2.

By Proposition 6.3, a transformation that combines **T1** then **T2** is also a fold operation. It is induced by the function $\xi = G\zeta \circ \eta_{Hz}$. Function ξ has type

$$\begin{array}{rcl} \xi & : & (P_1(H(z_1, z_2, z_3), H(z_1, z_2, z_3)) \to \\ & & H(R_1(z_1, z_2, z_3), R_2(z_1, z_2, z_3), R_3(z_1, z_2, z_3))) \quad \otimes \\ & & (P_2(H(z_1, z_2, z_3), H(z_1, z_2, z_3)) \to \\ & & H(R_1(z_1, z_2, z_3), R_2(z_1, z_2, z_3), R_3(z_1, z_2, z_3))) \end{array}$$

and with the constraints that $z_1 = u_1, z_2 = u_2$, and $z_3 = u_3$. As with η, ξ has two components ξ_1 and ξ_2. ξ_1 assembles the content for a `list_even` element from the content of an `even` element, assuming the latter already consists of, if any, a `list_odd` element. Likewise, ξ_2 assembles the content for a `list_odd` element from the content of an `odd` element, assuming the latter must already consist of a `list_even` element.

Now, we need not code ξ at all! The code for ξ will be generated, automatically, by a higher-order module that takes both η and ζ as input and produces ξ as its output. The higher-order module simply implements the equation $\xi = G\zeta \circ \eta_{Hz}$ (see Proposition 6.3). This is demonstrated by module type `COMPOSE_PG2GQ_2_1_3` in Figure 4, where it takes two natural transformations (with one of them with additional sharing constraints), and produces a natural transformation (with additional sharing constraints).

9. MODULAR XML TRANSFORMATIONS WITH ML MODULES

We have used Objective Caml to prototype XML transformations according to the principles developed in this paper. Objective Caml is a functional language in the ML family that supports both a polymorphic type scheme and a parametric module system [3]. Our prototype is highly modular, in the sense that parametric modules are heavily used to structure, and to parameterize, layers of categorical constructions: Functors, parametric content models and their fixed-points, arity-adjusting adjoint functors, natural transformations induced by arity-adjusting functors, and finally, fold/unfold functions derived by natural transformations. Previously we have also used highly modular ML code for generic validation of XML elements [10]. The ML prototype for transforming structural content is generic as well, as modules in the prototype are DTD-parameterized and strongly typed. Note that, for illustrating purpose, the code we showed in Figure 2 is a de-functorized and simplified version of the fully modular code.

```
(*--- O'Caml type defintions for parametric content models P, Q, R. ---*)

type ('a, 'b)     p1 = 'b option
type ('a, 'b)     p2 = 'a
type  'a          q1 = 'a option
type ('a, 'b, 'c) r1 = 'c list
type ('a, 'b, 'c) r2 = 'c list
type ('a, 'b, 'c) r3 = unit

(*--- P, Q, R are functors, and the following are their "map" functions. ---*)

let map_p1 (f1, f2)     w = match w with None -> None | Some b -> Some (f2 b)
let map_p2 (f1, f2)     w = f1 w
let map_q1  f           w = match w with None -> None | Some a -> Some (f a)
let map_r1 (f1, f2, f3) w = List.map f3 w
let map_r2 (f1, f2, f3) w = List.map f3 w
let map_r3 (f1, f2, f3) w = ()

(*--- O'Caml type definitions for XML element types. ---*)

type even = Even of (even, odd) p1
 and odd  = Odd  of (even, odd) p2

type succ = Succ of succ q1

type list_even = ListEven of (list_even, list_odd, one) r1
 and list_odd  = ListOdd  of (list_even, list_odd, one) r2
 and one       = One      of (list_even, list_odd, one) r3

(*--- s2t and t2u are fold functions for XML transformations,   ---*)
(*---     constrained by the arity-adjusting functors G and H.  ---*)
(*--- Note that functor G is inlined into the defintion of s2t. ---*)

type ('z1, 'z2, 'z3) h = Z1 of 'z1 | Z2 of 'z2
let map_h (f1, f2, f3) w = match w with Z1 a -> Z1 (f1 a) | Z2 b -> Z2 (f2 b)

let s2t ((eta1: ('y1, 'y1) p1 -> 'y1 q1), (eta2: ('y1, 'y1) p2 -> 'y1 q1)) =
    let rec even2succ (Even p1) = Succ (eta1 (map_p1 (even2succ, odd2succ) p1))
        and  odd2succ (Odd  p2) = Succ (eta2 (map_p2 (even2succ, odd2succ) p2))
 in
    (even2succ, odd2succ)

let t2u (zeta: ('z1, 'z2, 'z3) h q1 -> (('z1, 'z2, 'z3) r1, ('z1, 'z2, 'z3) r2, ('z1, 'z2, 'z3) r3) h) =
    let an_even_list   w = ListEven   w
 in let an_odd_list    w = ListOdd    w
 in let rec don't_care w = don't_care w
 in let rec succ2list (Succ q1) = map_h (an_even_list, an_odd_list, don't_care) (zeta (map_q1 succ2list q1))
 in
    succ2list

(*--- eta and zeta form the inductive bases of the fold functions;                 ---*)
(*--- f1, f2, and g are the transformations as required by T1 and T2 in Section 8. ---*)

let eta1 w = w
let eta2 w = Some w
let zeta w = match w with None                    -> Z1 []
                        | Some (Z1 (ListEven z1)) -> Z2 ((One ()) :: z1)
                        | Some (Z2 (ListOdd  z2)) -> Z1 ((One ()) :: z2)

let ((f1: even -> succ), (f2: odd -> succ))   = s2t (eta1, eta2)
let (g: succ -> (list_even, list_odd, one) h) = t2u zeta
```

Figure 2: De-functorized Objective Caml code for the example in Section 8. Note that type annotations are not needed; they are added for clarity purposes.

```
module type FUN1 =
sig
  type 'x1 t
  val map: ('x1 -> 'y1) -> 'x1 t -> 'y1 t
end

module type FUN2 =
sig
  type ('x1, 'x2) t
  val map: ('x1 -> 'y1) * ('x2 -> 'y2) ->
           ('x1, 'x2) t -> ('y1, 'y2) t
end

module type ATT2 =
sig
  type t1
  type t2
end

module type SYS2 =
sig
  module F1: FUN2
  module F2: FUN2
end

module type DTD2 =
sig
  module A: ATT2
  module S: SYS2
  type t1
  type t2

  module T1:
  sig
    type t = t1

    val up:   A.t1 * (t1, t2) S.F1.t -> t
    val down: t -> A.t1 * (t1, t2) S.F1.t
  end

  module T2:
  sig
    type t = t2

    val up:   A.t2 * (t1, t2) S.F2.t -> t
    val down: t -> A.t2 * (t1, t2) S.F2.t
  end
end
```

Figure 3: Module interfaces, in Objective Caml, for modular XML transformations. (Part 1 of 2)

Figures 3 and 4 show brief segments of the modular code. They contain the interfaces (called *module types* in Objective Caml) of the major modules that are involved in the layered constructions. The fold functions that map from a binary DTD to a unary DTD are named as `f1` and `f2` in module type `XF_2_1`. In turn, a module of type `XF_2_1` is returned by a module of type `FOLD_2_1`. As one can see in Figure 3, we use an m-ary type constructor in ML to model an m-ary functor. A set of m m-ary functor (c.f. module type `SYS2`) is needed to specify an XML DTD with a set of m m-ary parametric content models. One may wonder if this approach is practical as a DTD may contain up to a hundred element types, and as such, we need 100-ary (and up) type constructors. But as we have shown in our previous study [10], this is not a problem for the Objective Caml compiler.

The complete Objective Caml code of the prototype can be found at the following URL:

`http://www.iis.sinica.edu.tw/~trc/x_dot_ml.html`

10. IS IT PRACTICAL?

Is this framework of modular XML transformations ever practical? Can it apply to schema languages other than DTD, e.g., XML Schema [4] and Relax NG [2]. Before answering these criticisms, we would like to mention that the ML code in Figures 2, 3, and 4 is not necessarily to be written by human. Instead, the code can be automatically generated by user specifications. The specifications in turn are formulated using GUI-based tools that allow users to set up mappings from source DTDs to target DTDs, visually. See, e.g., [14, 23] for work on visual specifications of XML transformations. Our contribution in this paper is to set up a framework where multiple specifications can be efficiently combined and fused into one.

Our mapping of an XML document to an ML-typed value depends on XML DTD's 1-deterministic content model property[2]. That is, we take for granted that an XML document (with a DTD) will uniquely "parsed" into an ML value (of the corresponding ML data type). This ensures that an XML transformation specified by a DTD-to-DTD natural transformation contains no ambiguity. XML Schema, like XML DTD, maintains the 1-deterministic content model property but Relax NG does not. As a result, it will be more challenging to develop a similar transformation framework for Relax NG (whose content models allows ambiguity in the deviations of element sequences).

11. CONCLUSION

We have shown that certain XML document transformations can be modeled as a fold operation and as an unfold operation. Such a transformation is derived from a natural transformation from the source content model to the target content model. We also show that such XML document transformations are modular, in the sense that the composition of two XML document transformations is again characterized by a natural transformation. A prototype written

[2]Briefly, 1-deterministic content model demands that one look-ahead is always sufficient for the unique derivation of an element sequence from its content model.

```
module type ADJ_2_1 =
sig
  module F:
  sig
    module F1: FUN2
  end

  module G:
  sig
    module F1: FUN1
    module F2: FUN1
  end

  type ('x1, 'x2, 'y1) fx2y = ('x1, 'x2) F.F1.t -> 'y1
  type ('x1, 'x2, 'y1) x2gy = ('x1 -> 'y1 G.F1.t) * ('x2 -> 'y1 G.F2.t)

  val psi:    ('x1, 'x2, 'y1) fx2y -> ('x1, 'x2, 'y1) x2gy
  val varphi: ('x1, 'x2, 'y1) x2gy -> ('x1, 'x2, 'y1) fx2y
end

module type PG2GQ_2_1 =
sig
  module A: ATT2
  module P: SYS2
  module B: ATT1
  module Q: SYS1
  module X: ADJ_2_1

  module Eta: functor (T: sig  type t1  end) ->
  sig
    val f1: A.t1 * (T.t1 X.G.F1.t, T.t1 X.G.F2.t) P.F1.t -> (B.t1 * T.t1 Q.F1.t) X.G.F1.t
    val f2: A.t2 * (T.t1 X.G.F1.t, T.t1 X.G.F2.t) P.F2.t -> (B.t1 * T.t1 Q.F1.t) X.G.F2.t
  end
end

module type XF_2_1 =
sig
  module S: DTD2
  module T: DTD1
  module X: ADJ_2_1

  val f1: S.t1 -> T.t1 X.G.F1.t
  val f2: S.t2 -> T.t1 X.G.F2.t
end

module type FOLD_2_1 = functor (Source: DTD2) -> functor (Target: DTD1) ->
  functor (Nat: PG2GQ_2_1 with module A = Source.A and module P = Source.S
                          and module B = Target.A and module Q = Target.S) ->
  XF_2_1 with module S = Source and module T = Target and module X = Nat.X

module type COMPOSE_PG2GQ_2_1_3 = functor (NatN: PG2GQ_1_3) ->
  functor (NatM: PG2GQ_2_1 with module B = NatN.A and module Q = NatN.P) ->
  PG2GQ_2_3 with module A = NatM.A and module P = NatM.P and module B = NatN.B
             and module Q = NatN.Q and module X = ComposeAdj_2_1_3 (NatN.X)(NatM.X)
```

Figure 4: Module interfaces, in Objective Caml, for modular XML transformations. (Part 2 of 2)

in Objective ML has been constructed to experiment with this modular style of XML document transformations.

12. REFERENCES

[1] Document Object Model (DOM) Level 1 Specification (Second Edition). <http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>. W3C Working Draft, 29 September, 2000.

[2] RELAX NG Specification. <http://www.oasis-open.org/committees/relax-ng/spec-20011203.html>. Committee Specification, 3 December 2001.

[3] The Caml Language. <http://caml.inria.fr>.

[4] XML Schema Part 0: Primer. <http://www.w3.org/TR/xmlschema-0/>. W3C Recommendation, 2 May 2001.

[5] E. Akpotsui, V. Quint, and C. Roisin. Type modelling for document transformation in structured editing systems. *Mathematical and Computer Modelling*, 25(4):1–19, 1997.

[6] Roland Backhouse, Patrick Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *Advanced Functional Programming, 3rd International School*, pages 28–115, September 1999. Lecture Notes in Computer Science, Volume 1608.

[7] Richard Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer–Verlag, 1987.

[8] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice Hall, 1997.

[9] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data source. In *Proceedings of the Third International Workshop on the Web and Databases*, May 2000.

[10] Tyng–Ruey Chuang. Generic validation of structural content with parametric modules. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 98–109, September 2001.

[11] Mary Fernandez, Jerome Simeon, and Philip Wadler. An algebra for XML query. December 2000. <http://www.cs.bell-labs.com/who/wadler/papers/xalgebra-india/xalgebra-india.pdf>.

[12] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases*, 2000.

[13] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming*, September 2000.

[14] Eila Kuikka, Paul Leinonen, and Martti Penttonen. Towards automating of document structure transformations. In *2nd ACM International Conference on Document Engineering*, November 2002.

[15] Eila Kuikka and Martti Penttonen. Transformation of structured documents. Technical Report CS-95-46, Department of Computer Science, University of Waterloo, Canada, October 1995.

[16] Saunders Mac Lane. *Categories for The Working Mathematician.* Springer-Verlag, 1971.

[17] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334, 1986.

[18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and bared wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144, August 1991. Lecture Notes in Computer Science, Volume 523.

[19] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for xml transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22, 2000.

[20] Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Third International Workshop on Principles of Document Processing*, September 1996.

[21] Makoto Murata. Data models for document transformation and assembly. In *Workshop on Principles of Digital Document Processing*, March 1998.

[22] Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–17, June 1998.

[23] Emmanuel Pietriga, Jean-Yves Vion-Dury, and Vincent Quint. VXT: A visual approach to xml transformations. In *1st ACM International Conference on Document Engineering*, October 2001.

[24] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359, September 1989.

APPENDIX

A. PROOF

Let $\mathcal{A}, \mathcal{B}$ be two categories, and $F : \mathcal{A} \to \mathcal{B}$, $G : \mathcal{B} \to \mathcal{A}$ be two functors between them. F is a *left adjoint* of G (equivalently, G is a *right adjoint* of F) if there exist natural transformations φ and ψ such that $\varphi_{x,y}(h) = k$ and $\psi_{x,y}(k) = h$, where $h : x \to Gy$ is an arrow in $\mathcal{A}$, and $k : Fx \to y$ an arrow in $\mathcal{B}$. Furthermore, φ and ψ constitute an isomorphism between the arrows in $\mathcal{A}$ and the arrows in $\mathcal{B}$.

$$\varphi_{x,y} \circ \psi_{x,y} = \mathrm{id}_{Fx \to y}$$
$$\psi_{x,y} \circ \varphi_{x,y} = \mathrm{id}_{x \to Gy}$$

From the naturality of φ and ψ, one has the following equations

$$\begin{aligned} \varphi_{x,y}(h \circ f) &= \varphi_{x,y}(h) \circ Ff \\ \varphi_{x,y}(Gg \circ h) &= g \circ \varphi_{x,y}(h) \\ \psi_{x,y}(k \circ Ff) &= \psi_{x,y}(k) \circ f \\ \psi_{x,y}(g \circ k) &= Gg \circ \psi_{x,y}(k) \end{aligned}$$

where f is an arrow in $\mathcal{A}$ and g an arrow in $\mathcal{B}$. Furthermore, from the definitions of *universal arrows* unit and counit,

$$\mathrm{unit}_x : x \to GFx = \psi_{x,Fx}(F\ \mathrm{id})$$
$$\mathrm{counit}_y : FGy \to y = \varphi_{Gy,y}(G\ \mathrm{id})$$

one also derives the following equations

$$\begin{aligned} \varphi_{x,y}(h) &= \mathrm{counit}_y \circ Fh \\ \psi_{x,y}(k) &= Gk \circ \mathrm{unit}_x \end{aligned}$$

Given $s = Ps, t = Qt$, and two natural transformations $\eta : PG \to GQ$ and $\theta : FP \to QF$, we can show the following equivalence relation.

LEMMA A.1. $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$ if and only if $\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy})$ ⋄

Proof. We show here that "only if" direction. Proof for the other direction is similar.

Our proof is based on the following commutative diagram.

$$\begin{array}{ccc} PGy & \underset{P\ \mathrm{unit}_{Gy}}{\overset{PG\mathrm{counit}_y}{\rightleftarrows}} & PGFGy \\ \downarrow{\scriptstyle \eta_y} & & \downarrow{\scriptstyle \eta_{FGy}} \\ GQy & \underset{GQ\ \mathrm{counit}_y}{\longleftarrow} & GQFGy \end{array}$$

We first show that

$$\begin{array}{rcl} PG\mathrm{counit}_y \circ P\ \mathrm{unit}_{Gy} & = & P(G\mathrm{counit}_y \circ \mathrm{unit}_{Gy}) \\ & = & P(\psi_{Gy,y}(\mathrm{counit}_y)) \\ & = & P(\psi_{Gy,y}(\varphi_{Gy,y}(\mathrm{id}_{Gy}))) \\ & = & P(\mathrm{id}_{Gy}) = \mathrm{id}_{PGy} \end{array}$$

Since η is a natural transformation, we have

$$\eta_y \circ PG\mathrm{counit}_y = GQ\ \mathrm{counit}_y \circ \eta_{FGy}$$

implies

$$\eta_y \circ PG\mathrm{counit}_y \circ P\ \mathrm{unit}_{Gy} = GQ\ \mathrm{counit}_y \circ \eta_{FGy} \circ P\ \mathrm{unit}_{Gy}$$

which in turn implies

$$\eta_y = GQ\ \mathrm{counit}_y \circ \eta_{FGy} \circ P\ \mathrm{unit}_{Gy}$$

Therefore,

$$\begin{array}{rcl} \varphi_{PGy,Qy}(\eta_y) & = & \varphi_{PGy,Qy}(GQ\ \mathrm{counit}_y \circ \eta_{FGy} \circ P\ \mathrm{unit}_{Gy}) \\ & = & Q\ \mathrm{counit}_y \circ \varphi_{PGy,QFGy}(\eta_{FGy} \circ P\ \mathrm{unit}_{Gy}) \\ & = & Q\ \mathrm{counit}_y \circ \theta_{Gy} \end{array}$$

Because ψ and φ are inverse to each other, the above equation implies

$$\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy}).$$

Proof about the other direction is similar, and is based on the following commutative diagram.

$$\begin{array}{ccc} FPx & \overset{FP\ \mathrm{unit}_x}{\longrightarrow} & FPGFx \\ \downarrow{\scriptstyle \theta_x} & & \downarrow{\scriptstyle \theta_{GFx}} \\ QFx & \underset{Q\ \mathrm{counit}_{Fx}}{\overset{QF\mathrm{unit}_x}{\rightleftarrows}} & QFGFx \end{array}$$

◇

We also prove the following two lemmas.

Lemma A.2. $G(\theta_s \circ F\mathrm{down}_s) \circ \mathrm{unit}_s = \psi_{Ps,QFs}(\theta_s) \circ \mathrm{down}_s$ ◇

Proof. We know that $\psi_{Ps,QFs}(\theta_s) = G\theta_s \circ \mathrm{unit}_{Ps}$. It follows that

$$\begin{array}{rl} & \psi_{Ps,QFs}(\theta_s) \circ \mathrm{down}_s \\ = & G\theta_s \circ (\mathrm{unit}_{Ps} \circ \mathrm{down}_s) \\ = & G\theta_s \circ (GF\mathrm{down}_s \circ \mathrm{unit}_s) \\ = & G(\theta_s \circ F\mathrm{down}_s) \circ \mathrm{unit}_s \end{array}$$

◇

Lemma A.3. $GQ\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \psi_{Ps,QFs}(\theta_s) = \eta_t \circ P(\psi_{s,t}\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket)$ if $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$. ◇

Proof. Since $\theta_s = \varphi_{Ps,QFs}(\eta_{Fs} \circ P\ \mathrm{unit}_s)$ and φ and ψ are inverse to each other, we know that $\psi_{Ps,QFs}(\theta_s) = \eta_{Fs} \circ P\ \mathrm{unit}_s$.

It follows that

$$\begin{array}{rl} & GQ\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \psi_{Ps,QFs}(\theta_s) \\ = & GQ\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \eta_{Fs} \circ P\ \mathrm{unit}_s \\ = & \eta_t \circ PG\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ P\ \mathrm{unit}_s \\ = & \eta_t \circ P(G\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \mathrm{unit}_s) \\ = & \eta_t \circ P(\psi_{s,t}\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket) \end{array}$$

◇

We now prove the main result.

Proposition A.4. The following statements are all true.

1. $\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis = \psi_{s,t}\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket$ if $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$.
2. $\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis = \psi_{s,t}\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket$ if $\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy})$.
3. $\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket = \varphi_{s,t}\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis$ if $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$.
4. $\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket = \varphi_{s,t}\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis$ if $\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy})$.

◇

Proof. Note that $\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis = \psi_{s,t}\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket$ if and only if $\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket = \varphi_{s,t}\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis$. Note also that, by Lemma A.1, $\theta_x = \varphi_{Px,QFx}(\eta_{Fx} \circ P\ \mathrm{unit}_x)$ if and only if $\eta_y = \psi_{PGy,Qy}(Q\ \mathrm{counit}_y \circ \theta_{Gy})$. It follows that if one of the four statements is true, then all four statements are true. We now show the first statement to be true.

$$\begin{array}{rl} & \psi_{s,t}\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \\ = & G\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \mathrm{unit}_s \\ = & G\mathrm{up}_t \circ GQ\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ G(\theta_s \circ F\mathrm{down}_s) \circ \mathrm{unit}_s \\ & \quad \text{— Definition of unfold} \\ = & G\mathrm{up}_t \circ GQ\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket \circ \psi_{Ps,QFs}(\theta_s) \circ \mathrm{down}_s \\ & \quad \text{— Lemma A.2} \\ = & G\mathrm{up}_t \circ \eta_t \circ P(\psi_{s,t}\llbracket\theta_s \circ F\mathrm{down}_s\rrbracket) \circ \mathrm{down}_s \\ & \quad \text{— Lemma A.3} \end{array}$$

By the uniqueness of the fold operator, it follows that $\llparenthesis G\ \mathrm{up}_t \circ \eta_t \rrparenthesis = \psi_{s,t}\llbracket\theta_s \circ F\ \mathrm{down}_s\rrbracket$ ◇