On Modular Transformation of Structural Content

Tyng–Ruey Chuang　　Jan–Li Lin
Institute of Information Science
Academia Sinica
Nankang, Taipei 115
Taiwan

{trc, ljl}@iis.sinica.edu.tw

Outline

- XML: What and Why
- Mapping XML DTDs to ML Type Definitions
- Fold/Unfold and Natural Transformation
- Modularity of Fold/Unfold
- Dealing with Mis-matched Arities
- Modeling XML Transformations in ML
- Concluding Remark

XML: What and Why

- XML is an extensible markup language for tagging documents for their structural content.
- XML is extensible because each XML document can include a *Document Type Definition* (DTD) that specifies its own tagging rules.
- XML is for exchange of complex documents/datasets.

Document Type Definition

- An XML document is a tree of *elements*. An element consists of the start-tag `<name>`, the end-tag `</name>`, and a sequence of child elements in between.

- A DTD is a set of (mutually recursive) regular expression definitions of element type names. For an element type T, its defining regular expression specifies what element sequences are valid as children for elements of type T. The regular expression is called T's *content model*.

- *Well-formed*: the start-tags and end-tags are properly matched.

- *Valid*: the child sequence is derivable from the content model.

A Tidy Bookmark Folder: An Example

- The following XML document has a `folder` DTD. The DTD has two element types `folder` and `record`.

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

- This “tidy” DTD specifies that a record must not contain any element, and no folder is ever empty or contains just one folder.
- The above document is a valid XML document.

Map XML Element Types to ML Types (I)

Define ML type constructors for all the XML content model operators. Define XML element types using only these ML type constructors.

```
type ('a, 'b) alt = L of 'a | R of 'b               (*  "|"  *)
type ('a, 'b) seq = 'a * 'b                         (*  ","  *)
type 'a star = 'a list                              (*  "*"  *)
type 'a plus = One of 'a | More of 'a * 'a plus     (*  "+"  *)

type folder = Folder of ((record, (folder, record) alt star) seq,
                         (folder, (folder, record) alt plus) seq) alt
 and record = Record
```

Map XML Element Types to ML Data Types (II)

Abstract the right-hand-sides of the type equations into type constructors, and express the XML element types as simultaneous fixed points of these type constructors.

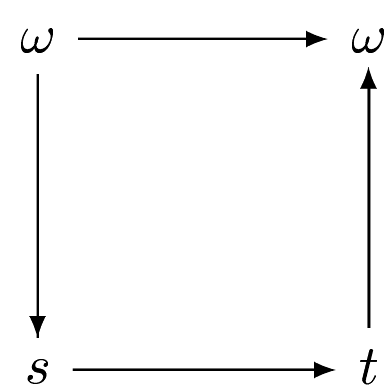
```
type ('a, 'b) f0 = (('b, ('a, 'b) alt star) seq,
                    ('a, ('a, 'b) alt plus) seq) alt
type ('a, 'b) f1 = unit

type folder = Folder of (folder, record) f0
 and record = Record of (folder, record) f1
```

We call type constructors `f0` and `f1` *parametric content models.*

The Big Picture

Let s and t be XML DTDs. Each also denotes the set of valid XML documents w.r.t. the DTD. Let ω be the set of all well-formed XML documents.

- The functions in $\omega \to \omega$ are “untyped”, while those in $s \to t$ are “typed”. Validation is a function in $\omega \to s$. (ICFP 2001)
- How to model and compose functions from s to t? (This Talk)



Some Notations

Let $s = (s_1, s_2, \ldots, s_n)$ denote a DTD s consisting of a tuple of element types $s_1, s_2, \ldots, s_n$, which are defined as the simultaneous fixed point of parametric content models $P = (P_1, P_2, \ldots, P_n)$. That is,

$$(s_1, s_2, \ldots, s_n) = (P_1(s_1, s_2, \ldots, s_n), P_2(s_1, s_2, \ldots, s_n), \ldots, P_n(s_1, s_2, \ldots, s_n))$$

We use $s = Ps$ to denote s as the fixed point of P.

Let $\mathrm{up}_s : Ps \to s$ and $\mathrm{down}_s : s \to Ps$ be the two mappings that together defines the identities

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

Paramertric Content Models are Functors

Define $Pf : Ps \to Pt$ for $f = (f_1, f_2, \ldots, f_n)$, where $f_i : s_i \to t_i$, 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 $P_i(f_1, f_2, \ldots, f_n)$ is the function that map value $P_i(v_1, v_2, \ldots, v_n)$ to value $P_i(f_1(v_1), f_2(v_2), \ldots, f_n(v_n))$.

Moreover,

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

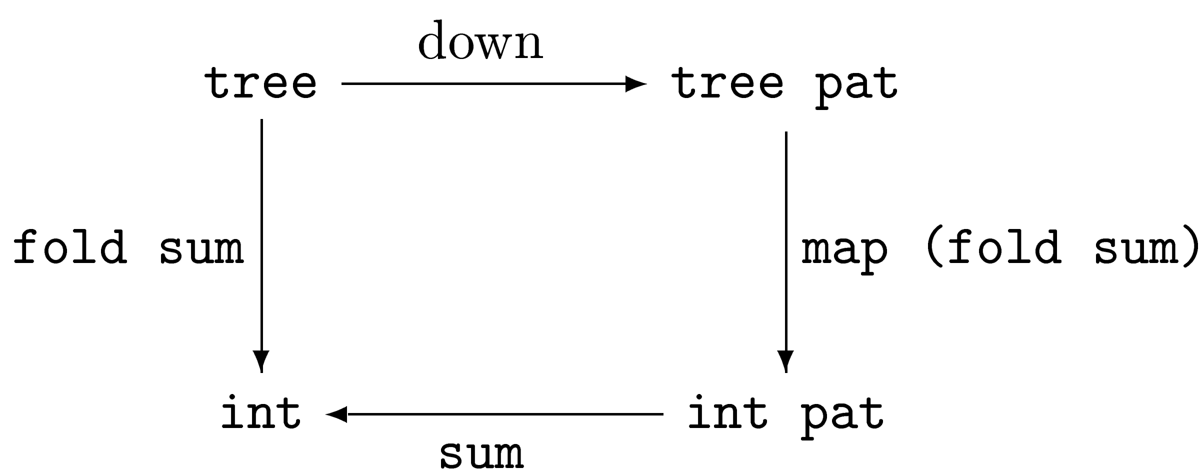
for all $f : s \to t$ and $g : t \to u$. P is a *functor*, categorical speaking.

Fold — An Example

```
type 'a pat = Nil | Node of 'a * 'a
let map f pat =
    match pat with Nil -> Nil | Node (x, y) -> Node (f x, f y)

type tree = Rec of tree pat
let up         pat  = Rec pat
let down (Rec pat) =     pat
let rec fold f tree = f (map (fold f) (down tree))

let sum pat = match pat with Nil -> 0 | Node (x, y) -> x + y + 1
let count tree = fold sum tree

let my_tree  = Rec (Node (Rec Nil, Rec (Node (Rec Nil, Rec Nil))))
let my_total = count my_tree
```

The Fold Diagram

```
map:   ('a -> 'b) -> 'a pat -> 'b pat
up:    tree pat -> tree
down:  tree -> tree pat
fold:  ('a pat -> 'a) -> tree -> 'a
sum:   int pat -> int
count: tree -> int
```

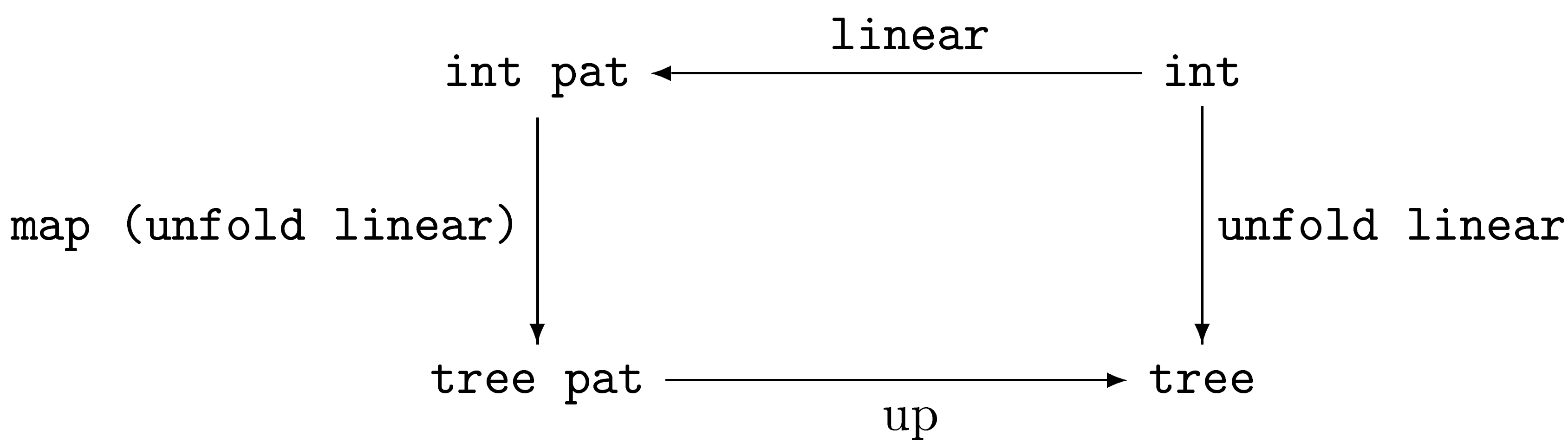


Unfold — An Example

```
type 'a pat = Nil | Node of 'a * 'a
let map f pat =
   match pat with Nil -> Nil | Node (x, y) -> Node (f x, f y)

type tree = Rec of tree pat
let up         pat  = Rec pat
let down (Rec pat) =     pat
let rec unfold g seed = up (map (unfold g) (g seed))

let linear n = if n<= 0 then Nil else Node (0, n - 1)
let skew   n = unfold linear n

let my_total = 2
let my_tree  = skew my_total
```

The Unfold Diagram

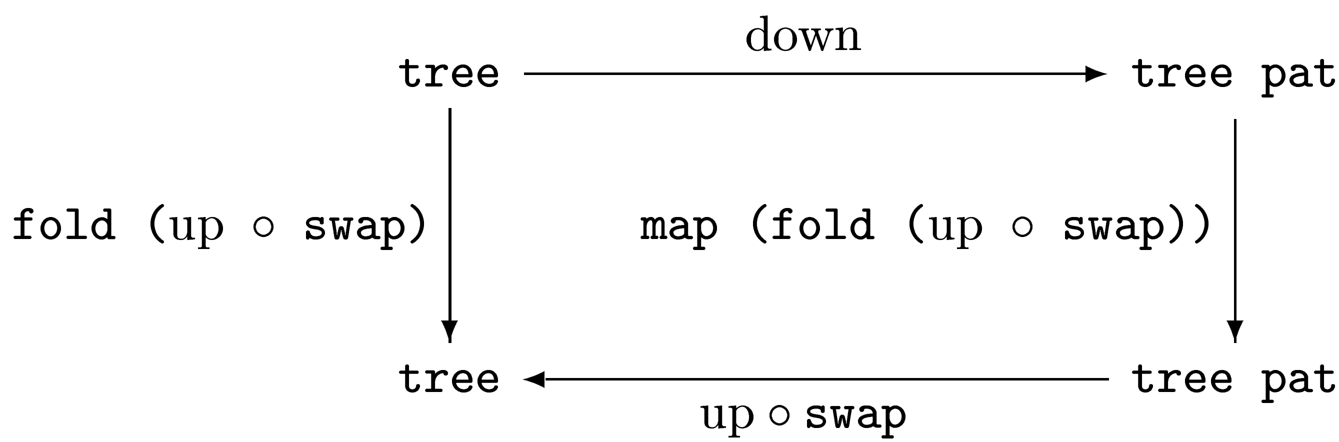
```
map:    ('a -> 'b) -> 'a pat -> 'b pat
up:     tree pat -> tree
down:   tree -> tree pat
unfold: ('a -> 'a pat) -> 'a -> tree
linear: int -> int pat
skew:   int -> tree
```

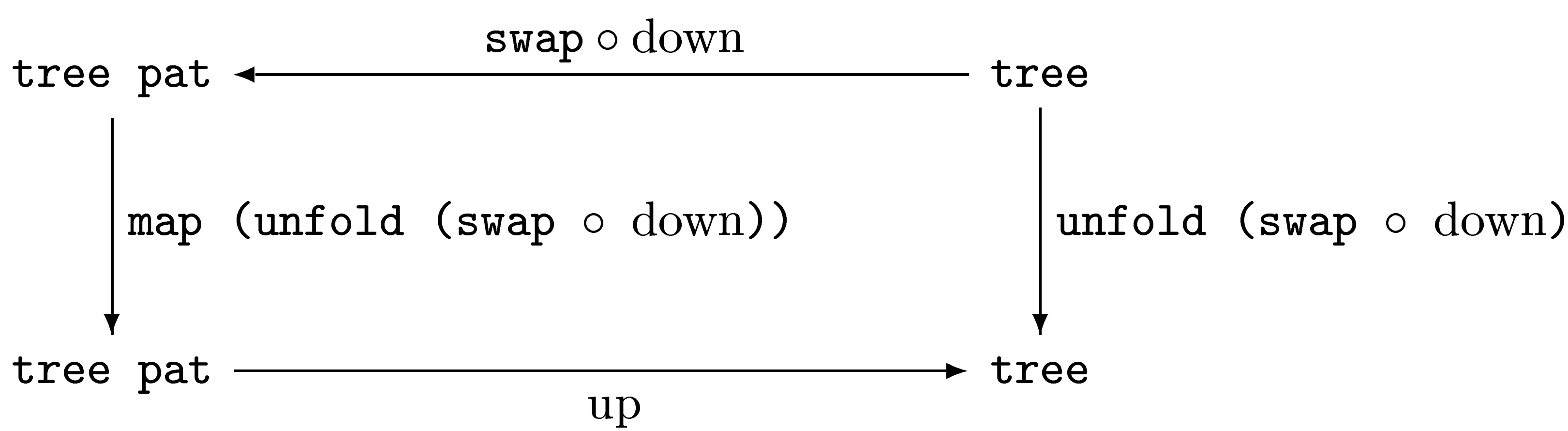


Fold or Unfold?

```
let rec   fold f tree = f  (map (  fold f) (down tree))
let rec unfold g seed = up (map (unfold g) (g    seed))

let swap pat = match pat with Nil->Nil | Node(x,y) -> Node(y,x)
let mirror_fold   tree =   fold (up   o swap) tree
let mirror_unfold tree = unfold (swap o down) tree

(* ---------------------------------------------------- *)

fold:          ('a pat -> 'a) -> tree -> 'a
unfold:        ('a -> 'a pat) -> 'a -> tree
swap:          'a pat -> 'a pat
up   o swap: tree pat -> tree
swap o down: tree -> tree pat
```

The Two Diagrams





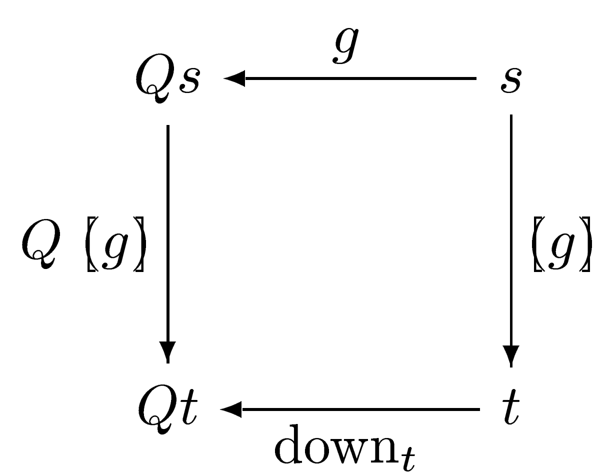
XML Document Transformation as Fold

Let DTDs $s = Ps$ and $t = Qt$ each defines 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 fold function if it is characterized by a reduction function $f : Pt \to t$ with the following commutative diagram:

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

XML Document Transformation as Unfold

A function from s to t is an unfold function if it is characterized by a generating function $g : s \to Qs$ with the following commutative diagram:

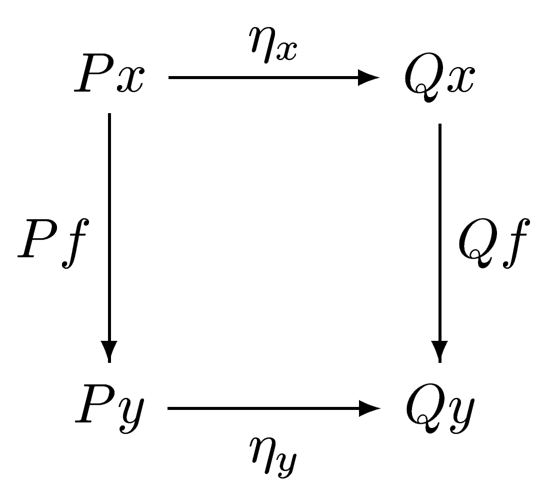


Natural Transformation

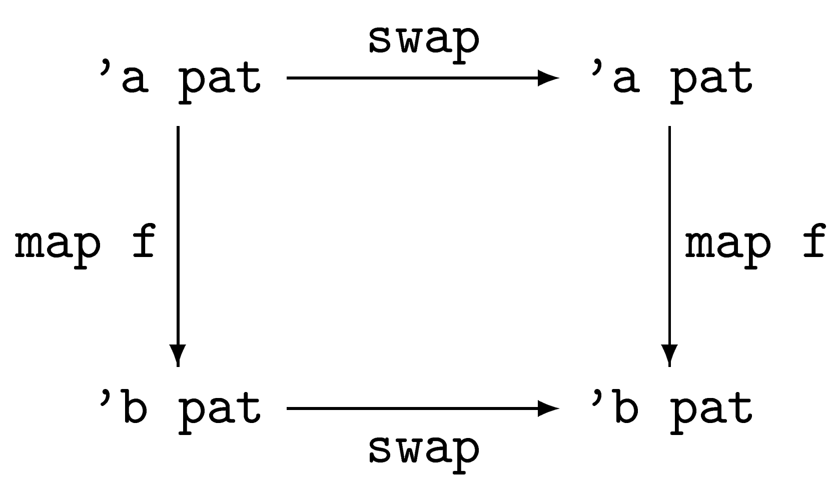
Let P and Q be two functors. A natural transformation $\eta : P \to Q$ is a collection of DTD-index functions that satisfies

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

for any DTD x and y, and for any function $f : x \to y$. That is, the following diagram commutes:



Function `swap` Is A Natural Transformation

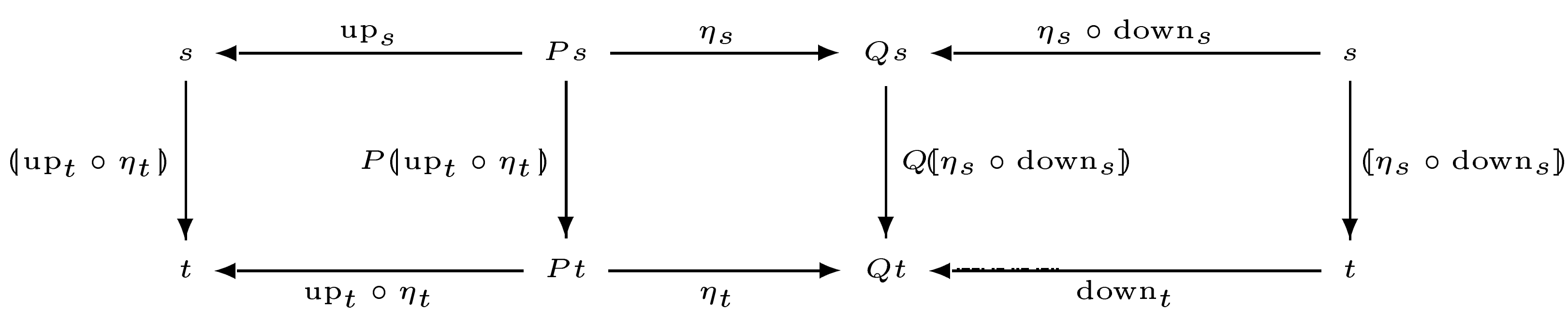


Fold/Unfold via Natural Transformation

A natural transformation $\eta : P \to Q$ defines two $s \to t$ functions:

- $(\!|\mathrm{up}_t \circ \eta_t|\!)$, this is a fold function;
- $[\!(\eta_s \circ \mathrm{down}_s)\!]$, this is an unfold function.

Furthermore, $(\!|\mathrm{up}_t \circ \eta_t|\!) = [\!(\eta_s \circ \mathrm{down}_s)\!]$.



The Two Mirrors Concide

```
mirror_fold =   fold (up   o swap)
            = unfold (swap o down)  = mirror_unfold
```

Modularity (I)

Let $s = Ps, t = Qt$, and $u = Ru$ be DTDs, and let $\eta : P \to Q$ and $\zeta : Q \to R$ be two natural transformations. Then

$$(\!|\mathrm{up}_u \circ \zeta_u|\!) \circ (\!|\mathrm{up}_t \circ \eta_t|\!) = (\!|\mathrm{up}_u \circ \zeta_u \circ \eta_u|\!)$$

That is, the composition of two folds is also a fold. Moreover,

$$\xi_x : P \to R = \zeta_x \circ \eta_x$$

is a natural transformation. That is, the resulting fold is again characterized by a natural transformation.

Modularity (II)

Similarly, for unfold, we have

$$[\!(\zeta_t \circ \mathrm{down}_t)\!] \circ [\!(\eta_s \circ \mathrm{down}_s)\!] = [\!(\zeta_s \circ \eta_s \circ \mathrm{down}_s)\!]$$

That is, the composition of two unfolds is also an unfold. Again,

$$\xi_x : P \to R = \zeta_x \circ \eta_x$$

is a natural transformation, and the resulting fold is characterized by a natural transformation.

Modeling XML Transformations in ML

- A ML codification of part of the category theory.
- Layers of categorical constructions are systematically mapped to layers of higher-order ML modules.
- The modules are of fixed arities, but are parameterized by DTD expressions.
- Issues of scalability and programming supports: An XML DTD may define 100-plus element types.

Concluding Remark

- ML is very useful in *modeling* XML processing in a modular way.
- Standard results from category theory are very helpful.
- Further generalization is possible: Just extend the index-set mapping $\sigma : M \to N$ from a function to a relation.
- Natural transformations can be too restrictive: They suffice to fuse two folds into one, but not necessarily all fusable folds must derived from natural transformations.