Generic Validation of Structural Content with Parametric Modules

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

ABSTRACT
In this paper, we demonstrate a natural mapping from element types of XML to module expressions of ML-like programming languages. The mapping is inductive, and the definitions of common XML operations can be derived as the module expressions are constructed. We show how to derive, in a generic way, the validation function, which checks an XML document for conformance to the content model specified by its DTD (Document Type Definition). One can view the validation function as giving types to XML elements, and the validation procedure a pre-requrement for typeful XML programming in ML.

Our mapping of XML element types to ML module expressions uses the parametric module facility of ML in some contrived way. For example, in validating WML (WAP Markup Language) documents, we need to use 36-ary type constructors, as well as higher-order modules that take in as many as 17 modules as input. That one can systematically model XML DTD at the module level suggests ML-like languages are suitable for type-safe prototyping of DTD-aware XML applications.

1. INTRODUCTION & MOTIVATION
XML (eXtensible Markup Language) is language for tagging documents for their structural content [2]. A XML document is tagged into a tree of nested elements. XML is extensible because each XML document can include a DTD (Document Type Definition) which lists the tags of the elements and specifies the tagging constraints. A central concept in XML document processing is validation. A XML document is valid if its content is tagged with the constraints specified by its DTD. A XML document is well-formed if each of its element is enclosed with matching start-tag and end-tag. A well-formed XML document is not necessarily valid.

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

The DTD in the above XML document models the structure where a record must contain no other element, and no folder is empty or contains just another folder. One may think of it modeling a tidy bookmark file. Of the following three elements, f3 is valid, but items f1 and f2 are not.

```xml
f1 <folder></folder>
f2 <folder><folder><record></record></folder></folder>
f3 <folder><folder><record></record></folder><record></record></folder>
```

Note that <record/> is a shorthand for <record></record>. The tag sequence <record><folder></record></folder> is an example of not-well-formedness.

To simplify discussion, we may say that each element type in the DTD is specified by its element content model (i.e., its tagging constraint) which is an unambiguous regular expression with element type names as symbols. The content model of an element type specifies what element sequences are allowed as the children of the element. Naturally, when coding XML programs, one need to map the element types in a DTD to the corresponding data types in the source programming language. A further requirement of the mapping is that content validation is translated into type correctness in the programming language, so that well-typed programs will always produce valid XML elements. Note that this goes beyond what is required of the so-called "validating..."
XML processors", which need only report violations of element content models in the input XML document but need not impose restrictions on the output.

There have been several directions in programming language support for writing XML applications. We can classify them into the following three categories.

**ADT for well-formed elements.** Abstract data types and the accompanying library routines are designed to traverse and transform well-formed XML elements. The XML data is assumed to be validated in a separate phase, or its validation is a separate issue and may not even be required. Examples in this category include standard XML API in C++, Java, or other languages (e.g., Document Object Model, DOM [1]) and a combinator approach to writing XML processing functional programs [3, 18].

**Type translation of DTD.** A strongly typed language is used for XML programming, and the type system of the language is used to embed DTDs. The embedding is complete (every element type has a corresponding data type in the embedding language) and sound (an expression of the embedding language evaluates to a valid XML element if the expression is well-typed in the language). Examples in this category include HaXml [3, 18] and XM Lambd a [14]. If the strongly typed language is statically typed, then the soundness proof is done by the type checker at compile-time. Hence no type-correct program will produce invalid XML elements. One can also use constraint-based languages or logic programming languages to encode XML content models in a similar way [19]. The type translation approach is not completely satisfactory for two reasons. One is that the type translation may not be systematic and can be tedious if done manually. The other inconvenience is that code for generic XML processing operations need to be rewritten for every DTD because they are translated into different types. XML content validation, which checks well-formed XML documents for conformance to their DTDs, is such a generic operation.

**Native language support of DTD.** New languages are being designed with built-in XML support to help build XML-related applications. XDuce is a functional language with regular expression types, so as to allow direct representations of DTDs and processing of valid elements [10, 11]. Expressions in the language are evaluated to valid XML elements, but variables must be annotated with their element types. The concept of validation is built into the language as type correctness, and programs are type-checked at compile-time. XDuce also provides regular expression patterns which further help write concise XML programs. XDuce, however, is currently a first-order and monomorphic language, and lacks some language features (e.g., a module system).

In this paper, we show how to use parametric modules in ML-like languages to write XML-supporting program modules that are both expressive and generic. It is expressive because all XML DTDs can be constructed from the provided parametric modules. It is generic because common operations, including the validation function, are automatically generated. As such, our approach has the advantages of both the type translation approach and the native DTD support approach, but without their disadvantages. There is no need to recode generic operations, and no need to design new language.

2. AN ILLUSTRATING EXAMPLE

For the tidy bookmark example described in Section 1, the following is the actual code we write in Objective Caml to specify the DTD, and to produce the validation functions for the two element types in the DTD.

```caml
module BookmarkTag =
  struct
    type ('x0, 'x1) t = Folder of 'x0 | Record of 'x1
    let map (f0, f1) t = ...
  end

module TidySys =
  struct
    module F0 = Alt(Seq(P1)(Star(Alt(P0)(P1))))
    (Seq(P0)(Plus(Alt(P0)(P1))))
    module F1 = Empty
    module Tag = BookmarkTag
  end

module TidyDtd = Mu(TidySys)
```

In the above, module TidySys contains two modules F0 and F1, which are translations, word by word, in Objective Caml module language the XML element type declarations of folder and record. The higher-order module Alt is for "|", Seq for ".", Star for "*", and Plus for "+". Ideally, we would like to define the two XML element types as two mutually recursive ML modules T0 and T1 as the following.

```caml
module T0 = Alt(Seq(T1)(Star(Alt(T0)(T1))))
(Seq(T0)(Plus(Alt(T0)(T1))))
and T1 = Empty
```

But Objective Caml, as most ML-like languages, does not support recursive modules. Instead we use two "place holder" modules P0 and P1 as the two parameters to higher-order modules (Alt, Seq, etc.), and use another higher-order module Mu (pronounced as μ) to derive the two simultaneous fixed points.

Module TidyDtd contains

- module U, which defines the type for well-formed elements;
- module V, which contains modules T0 and T1 that each defines the type for valid folder and record elements, respectively;
- functions validate and forget, which provide mappings between well-formed elements and valid elements.
It also defines exception Invalid, which may be raised by function validate. Note that the following equations always hold:

\[
\begin{align*}
\text{forget} \circ \text{validate} &= \text{id}, \quad \text{(may raise exception)} \\
\text{validate} \circ \text{forget} &= \text{id}
\end{align*}
\]

The sample element f3 as shown in Section 1 can now be defined and validated by the following Objective Caml code (f3.u is well-formed and f3.v is valid):¹

\[
\begin{align*}
\text{let } f3\_u &= \text{folder} [\text{folder} [\text{record} []]; \text{record} []] \\
\text{let } f3\_v &= \text{TidyDtd.\text{validate}} f3\_u
\end{align*}
\]

In addition, the valid element returned by the validation function is parsed and typed in the sense that all of its substructures are given specific types and can be extracted by using ML pattern-matching.

In this paper, we will use the above example to explain the idea and describe the construction. However, the idea and the construction can be systematically applied to DTDs with \(n\) element types. One need just to define a \(n\)-ary fixed point module \(\mu_n\), that will take a system of \(n\) \(n\)-ary higher-order modules \(F_0, F_1, \ldots, F_{n-1}\), and produce the simultaneous fixed points. The definition of \(\mu_n\) is symmetric and is similar to \(\mu_2\). We will later use WML (a markup language for wireless applications whose DTD defines 35 element types) as a benchmarking example to show the effectiveness of our approach.

## 3. GENERIC PROGRAMMING WITH PARAMETRIC MODULES

The XML element types in the folder example can be translated into Objective Caml using a series of type definitions as shown below.

\[
\begin{align*}
\text{type } ('a, 'b) \text{ alt} &= \text{L of } 'a \mid \text{R of } 'b \\
\text{type } ('a, 'b) \text{ seq} &= 'a * 'b \\
\text{type } 'a \text{ star} &= 'a \text{ list} \\
\text{type } 'a \text{ plus} &= \text{One of } 'a \mid \text{More of } 'a * 'a \text{ plus} \\
\text{type } \text{folder} &= \text{Folder of} \nonumber \\
&\quad ((\text{record}, (\text{folder}, \text{record}) \text{ alt star}) \text{ seq}, \nonumber \\
&\quad (\text{folder}, (\text{folder}, \text{record}) \text{ alt plus}) \text{ seq}) \text{ alt} \\
\text{and } \text{record} &= \text{Record}
\end{align*}
\]

One can abstract the right-hand-sides of the type equations for \text{folder} and \text{record} into two binary type constructors \text{f0} and \text{f1}, and view \text{folder} and \text{record} as the least fixed points of \text{f0} and \text{f1}.

¹Functions \text{folder} and \text{record} are syntactic sugars, and can be defined by

\[
\begin{align*}
\text{let } \text{folder ulist} &= \text{BookmarkTag.Folder} \\
&\quad (\text{TidyDtd.U.\text{up ulist}}) \\
\text{let } \text{record ulist} &= \text{BookmarkTag.Record} \\
&\quad (\text{TidyDtd.U.\text{up ulist}})
\end{align*}
\]

\[
\begin{align*}
\text{type } ('a, 'b) \text{ f0} &= (('b, ('a, 'b) \text{ alt star}) \text{ seq}, \nonumber \\
&\quad ('a, ('a, 'b) \text{ alt plus}) \text{ seq}) \text{ alt} \\
\text{type } ('a, 'b) \text{ f1} &= \text{unit} \\
\text{type } \text{folder} &= \text{Folder of} (\text{folder}, \text{record}) \text{ f0} \\
\text{and } \text{record} &= \text{Record of} (\text{folder}, \text{record}) \text{ f1}
\end{align*}
\]

One can further rewrite \text{f0} and \text{f1} using the two projection functions \text{p0} and \text{p1}, and the empty type constructor.

\[
\begin{align*}
\text{type } ('a, 'b) \text{ p0} &= 'a \\
\text{type } ('a, 'b) \text{ p1} &= 'b \\
\text{type } ('a, 'b) \text{ empty} &= \text{unit} \\
\text{type } ('a, 'b) \text{ f0} &= \nonumber \\
&\quad ((('a, 'b)\text{p1}, (('a, 'b)\text{p0}, ('a, 'b)\text{p1}) \text{ alt star}) \text{ seq}, \nonumber \\
&\quad ((('a, 'b)\text{p0}, (('a, 'b)\text{p0}, ('a, 'b)\text{p1}) \text{ alt plus}) \text{ seq}) \text{ alt} \\
\text{type } ('a, 'b) \text{ f1} &= ('a, 'b) \text{ empty}
\end{align*}
\]

At this point, it is clear that one can program in the module level, and define \text{f0} and \text{f1} as two module expressions using a predefined set of constant modules (for \text{p0}, \text{p1}, and \text{empty}), unary parametric modules (for \text{star} and \text{plus}), and binary parametric modules (for \text{alt} and \text{seq}). This is shown in Figure 1 where we also define the map function, inductively.

All XML element types can be defined using a fixed set of parametric modules.

We may say that modules \text{F0} and \text{F1} are objects in a functor category where each object has a type constructor \(t\) to map types to types, and a function \(\text{map}\) to map typed functions to typed functions. Parametric modules like \text{Plus} are arrows in the functor category, i.e., natural transformations. We view this definition of the map function a generic one, as each map instance is inductively indexed by its governing type expression. We will later show definitions of other generic values that are used in the definition of the validation function (which itself is generic as well).

## 4. PARAMETRIC CONTENT MODELS AND SIMULTANEOUS FIXED POINTS

In Figure 1, modules \text{F0} and \text{F1} each defines a binary type constructor \(t\), and the the two type constructors are used together to mutually define types \text{folder} and \text{record}. The code is reproduced below.

\[
\begin{align*}
\text{module } \text{F0}: \text{FUN} &= \text{Alt} (\text{Seq}(\text{P1})(\text{Star} (\text{Alt}(\text{PO}(\text{PO}))))))) \\
&\quad (\text{Seq}(\text{PO}))(\text{Plus} (\text{Alt}(\text{PO}(\text{PO}))))))) \\
\text{module } \text{F1}: \text{FUN} &= \text{Empty} \\
\text{type } \text{folder} &= \text{Folder of} (\text{folder}, \text{record}) \text{ \text{F0}.t} \\
&\quad \text{and } \text{record} &= \text{Record of} (\text{folder}, \text{record}) \text{ \text{F1}.t}
\end{align*}
\]

The type constructors \text{F0}.t and \text{F1}.t are parameteric content models in the sense that each maps a tuple of type instances to a content model. For example, given type instances \text{folder} and \text{record}, the type expression (folder, record) \text{F0}.t expands to

\[
\text{((record, (folder, record) \text{ alt star}) \text{ seq},}
\]

³
module type FUN =

  sig
    type ('a, 'b) t
    val map : ('a -> 'x) * ('b -> 'y) -> ('a, 'b) t -> ('x, 'y) t
  end

module type F2F = functor (F : FUN) -> FUN
module type F2F2F = functor (F0 : FUN) ->
                         functor (F1 : FUN) -> FUN

module Empty : FUN =

  struct
    type ('a, 'b) t = ()
    let map (f, g) t = ()
  end

module P0 : FUN =

  struct
    type ('a, 'b) t = 'a
    let map (f, g) t = f t
  end

module Plus : F2F =

  functor (F : FUN) ->

  struct
    type ('a, 'b) t =
      One of ('a, 'b) F.t
    | More of ('a, 'b) F.t * ('a, 'b) t
    let rec map (f, g) t =
      match t with
      One s -> One (F.map (f, g) s)
    | More (v, w) ->
      More (F.map (f, g) v, map (f, g) w)
  end

module Seq : F2F2F =

  functor (F0 : FUN) ->
                         functor (F1 : FUN) ->

  struct
    type ('a, 'b) t =
      (F0.map (f, g) (u, v),
       F1.map (f, g) v)
  end

module P1: FUN = ...
module Star: F2F = ...
module Alt: F2F2F = ...

module F0 : FUN =

  Alt(Seq(P1)(Star(Alt(P0)(P1))))
  (Seq(P0)(Plus(Alt(P0)(P1))))

module F1: FUN = Empty

  type folder = Folder of (folder, record) F0.t
  and record = Record of (folder, record) F1.t

Figure 1: Inductive definitions of XML element types using parametric modules.

Note: Module type annotations can be, and often are, omitted. We can take out the ":: F2F" part in "module Plus: F2F = ... " and at the same time expose the implementation of module Plus. The annotations are added for clarity and type-checking purposes.

which is exactly the XML content model for element type folder.

The main idea is to use type constructors as parametric content models, and view XML element types as simultaneous fixed points of a set of parametric content models. This viewpoint helps us develop primitive functions that are abstract and applicable to different content models (that is, the primitives are polymorphic). One of these primitives is the simultaneous induction operator — the fold function. We will later show that the validation procedure can be defined by using the fold function.

We then model two recursively defined XML element types by two interdependent ML modules T0 and T1. Their signatures are the following.

module T0:

  sig
    type ('x0, 'x1) cm
    type t
    val up : (T0.t, T1.t) cm -> T0.t
    val down T0.t -> (T0.t, T1.t) cm
  end

and

module T1:

  sig
    type ('x0, 'x1) cm
    type t
    val up : (T0.t, T1.t) cm -> T1.t
    val down T1.t -> (T0.t, T1.t) cm
  end

In the above, type constructor ('x0, 'x1) cm is for the parametric content model, and type t is for the element type. Functions up and down map between an element and its content model, and together define their equivalence:

\[ \text{down} \circ \text{up} = \text{id} \]
\[ \text{up} \circ \text{down} = \text{id} \]

Note that the above mutually defined signatures are not allowed in Objective Caml (as in most ML-like languages). However, one can use both auxiliary type names and additional type sharing constraints to overcome the problem. We can define a higher-order module MuValid that derives modules T0 and T1, when given a module that specifies the corresponding parametric content models and the tag set, see Figure 2. In Figure 2, modules F0 and F0 of the input module S specify the parametric content models, and module Tag specifies the tag set.

Note that, in the module returned by MuValid, the type for all valid elements is simply defined as the disjoint sum of type T0.t and type T1.t:
Also note that the simultaneous fold function has type

```
val fold: (('a, 'b) T0.cm -> 'a) *
(('a, 'b) T1.cm -> 'b) ->
(T0.t -> 'a) * (T1.t -> 'b)
```

Function fold returns with two reduction functions (whose types are T0.t -> 'a and T1.t -> 'b) if given two properly typed induction functions as bases (whose types are ('a, 'b) T0.cm -> 'a and ('a, 'b) T1.cm -> 'b).

Similarly, a higher-order module MuWF can be defined to derive a module for all well-formed elements; see Figure 3. In module MuWF, type constructor ('x0, 'x1) cm — the parametric content model for well-formed elements — is defined as a list of tagged values:

```
type ('x0, 'x1) cm = ('x0, 'x1) Tag.t list
```

and type u — the type for well-formed elements — is defined as the fixed point of the parametric content model cm:

```
type u = U of (u, u) cm
```

Note as well that type of all well-formed elements, type t, is defined as the disjoint sum of u and u, representing elements with two distinct tags. The definition of the simultaneous fold function is the same as that in module MuValid.

In Figure 3, there are several functions in module U2V and V2U that are given their types but are left undefined. They are used to specify functions validate and forget. Function validate maps a well-formed element to a valid element, while forget is the inverse function. Let us look at functions cm0 and cm1 in module U2V first. Their types are the following

```
val cm0: (V.T0.t, V.T1.t) U.cm ->
(V.T0.t, V.T1.t) V.T0.cm
val cm1: (V.T0.t, V.T1.t) U.cm ->
(V.T0.t, V.T1.t) V.T1.cm
```

Function cm0 maps a well-formed content, whose constituting parts are valid elements already, into a valid content. If function cm0 is composed with function V.T0.up, one gets a function that returns a valid element of type V.T0.t as result (we use $ as the function composition operator):

```
V.T0.up $ cm0: (V.T0.t, V.T1.t) U.cm -> V.T0.t
V.T1.up $ cm1: (V.T0.t, V.T1.t) U.cm -> V.T1.t
```

Given these two functions as the inductive bases to the simultaneous fold function, one derives the validation functions for elements of types V.T0.t and V.T1.t.

```
module type TAG =
sig
  type ('x0, 'x1) t
  val map: ('x0 -> 'y0) * ('x1 -> 'y1) ->
    ('x0, 'x1) t -> ('y0, 'y1) t
end

module type SYS =
sig
  module F0: FUN
  module F1: FUN
  module Tag: TAG
end

module MuValid = functor (S: SYS) ->
struct
  module Tag = S.Tag
  type t0 = V0 of (t0, t1) S.F0.t
  and t1 = V1 of (t0, t1) S.F1.t
  type t = (t0, t1) Tag.t

  module T0 =
  struct
    type ('x0, 'x1) cm = ('x0, 'x1) S.F0.cm
    let map = S.F0.map
    type t = t0
    let up cm = V0 cm
    let down (V0 cm) = cm
  end

  module T1 =
  struct
    type ('x0, 'x1) cm = ('x0, 'x1) S.F1.cm
    let map = S.F1.map
    type t = t1
    let up cm = V0 cm
    let down (V0 cm) = cm
  end

  let fold (f0, f1) =
    let rec fold0 x = f0 (T0.map (fold0, fold1) (T0.down x))
    and fold1 x = f1 (T1.map (fold0, fold1) (T1.down x))
    in
      (fold0, fold1)
  end
```

Figure 2: Module MuValid derives element types as simultaneous fixed points of a set of parametric content models.
module MuWF = functor (T: TAG) ->
struct
  module Tag = T
  type ('x0, 'x1) cm = ('x0, 'x1) Tag.t list
  let map fg = List.map (Tag.map fg)
  type u = U of (u, u) cm
  type t = (u, u) Tag.t
end

module Mu = functor (S: SYS) ->
struct
  module Sys = S
  module U = MuWF(Sys.Tag)
  module V = MuValid(Sys)
  exception Invalid
end

module U2V =
struct
  let cm0: (V.T0.t, V.T1.t) U.cm ->
  (V.T0.t, V.T1.t) V.T0.cm = ...

  let cm1: (V.T0.t, V.T1.t) U.cm ->
  (V.T0.t, V.T1.t) V.T1.cm = ...

  let (t0, t1): (U.u -> V.T0.t) * (U.u -> V.T1.t) =
  U.fold (V.T0.up $ cm0, V.T1.up $ cm1)
  let t: U.t -> V.t = Sys.Tag.map (t0, t1)
end

module V2U =
struct
  let cm0: (U.u, U.u) V.T0.cm ->
  (U.u, U.u) U.cm = ...

  let cm1: (U.u, U.u) V.T1.cm ->
  (U.u, U.u) U.cm = ...

  let (t0, t1): (V.T0.t -> U.u) * (V.T1.t -> U.u) =
  V.fold (U.up $ cm0, U.up $ cm1)
  let t: V.t -> U.t = Sys.Tag.map (t0, t1)
end

let validate = U2V.t
let forget = V2U.t

module type FUN =
sig
  type ('x0, 'x1) t
  val map: ('x0 -> 'y0) * ('x1 -> 'y1) ->
  ('x0, 'y0) t -> ('y0, 'y1) t

  val nullable: bool
  val first: Natset.t
end

Figure 3: Module MuWF derives the type for well-formed elements. Module Mu uses simultaneous fold to define the validation function.

Note: Type annotations for functions are added for clarity purpose.

Recall that the types for all well-formed elements and all valid elements are defined by

let U.t = (U.u, U.u) Tag.t
let V.t = (V.T0.t, V.T1.t) Tag.t

It follows that the validation function is defined by

let validate = Tag.map $
  U.fold (V.T0.up $ cm0, V.T1.up $ cm1)

As shown in Figure 3, one can define function forget in a similar way. It remains to be shown how functions like cm0 and cm1 are defined for all content models. This is shown next.

5. GENERIC VALIDATION OF CONTENT MODELS

Recall that, in Figure 1, a map function is defined in a generic way for any module with signature FUN, as long as the module is generated with the predefined set of parametric modules (Empty, P0, P1, Star, etc.). The validation and forgetting functions can be defined in a generic way as well. First we define the validation functions for the inductive bases. The validation function for any other content model can then be derived, automatically, as module expressions for the content are built.

There are two remaining details. The first is that at the time of building the content model, one does not have access to the tag module. This tag module is of signature TAG, and defines the variant data type for tagging elements (e.g., module BookmarkTag in Section 2). Therefore the validation and forgetting functions must reside in a higher-order module that takes in a TAG module as input.

One need also to maintain a nullable condition and a first set of element tags. A content model is nullable if it accepts the empty element sequence. The first set contains all tags that can appear at the first position of a valid sequence. It can be used to check if a content model is ambiguous, e.g., when the first sets of the two input modules to Alt overlap. When combined with a lookahead tag, it is used to implement a non-backtracking validation procedure as well. (More on this in Section 8.) Both nullable and first are generic values. The module signature FUN for parametric content model now consists of the following components.

module type FUN =
sig
  type ('x0, 'x1) t
  val map: ('x0 -> 'y0) * ('x1 -> 'y1) ->
  ('x0, 'y0) t -> ('y0, 'y1) t

  val nullable: bool
  val first: Natset.t
end
module Content: functor (T: TAG) ->
sig
  val validate: ('x0, 'x1) T.t list ->
    (('x0, 'x1) t * ('x0, 'x1) T.t list) Option.t
  val forget: ('x0, 'x1) t -> ('x0, 'x1) T.t list list end

Function validate takes a list of tagged values and turns it into a value of content model followed by the remaining list. Note that the type for the input, ('x0, 'x1) T.t list, is the same as the content model of well-formed element if the two share the same tag set. Figure 4 illustrates the construction by showing the implementations of modules P0 and Star.

The validation and forgetting functions are wrapped in module Content. The definition of Content is inductive: It depends on the Content module in the input module F (see, e.g., the module expression CM = F.Content(T) in module Star). We can view this as constituting a generic definition of the validation function, as each instance is systematically generated by its module expression. As evident in module Star, we adopt the longest prefix matching rule in validating the input element sequence against the "*" content model. This longest prefix matching rule is indeed required by XML. Validation functions for other modules, i.e., Empty, P0, P1, Plus, Seq, and Alt, can be similarly defined and are omitted here.

Now we return to Figure 3 to complete the definitions of functions cm0 and cm1 in modules U2V and V2U. They are defined as the following.

module U2V =
  struct
    module CM0 = Sys.F0.Content(Sys.Tag)
    let cm0 ulist =
      match CM0.validate ulist with
        Some (v, []) -> v
      | _ -> raise Invalid

  ...
end

module V2U =
  struct
    module CM0 = Sys.F0.Content(Sys.Tag)
    let cm0 = CM0.forget
    ...
end

Function cm0 in module U2V need to validate the input sequence of tagged value with the content model of element type V.T0.t, using the current tag set. This can be accomplished by using the validation function in module Sys.F0.Content(Sys.Tag). The only difference is that, if there remains a non-empty sequence after a validated (longest) prefix, the entire sequence is not valid with respect to the content model V.T0.t.

module P0: FUN =
  struct
    type ('x0, 'x1) t = 'x0
    let nullable = false
    let first = Natset.of_list [0]

  module Content = functor (T: TAG) ->
    struct
      let validate ulist =
        match ulist with
          [] -> None
        | h::t -> T.fold ((fun x -> Some (x, t)), (fun x -> None)) h
          (* if success, return the untagged value along with the remaining list; otherwise returns None. *)

      let forget a = [T.x0 a] (* Tag with the first variant of type T.t *)
    end

end

module Star: F2F = functor (F: FUN) ->
  struct
    type ('x0, 'x1) t = ('x0, 'x1) F.t List.t
    let nullable = true
    let first = F.first

  module Content = functor (T: TAG) ->
    struct
      module CM = F.Content(T)
      let rec validate ulist =
        match ulist with
          [] -> Some ([], ulist)
        | h::t -> if ... h in first ...
          then match CM.validate ulist with
            Some (u, t) ->
              (match validate t with
                Some (us, s) -> Some (u::us, s)
              | None -> Some ([u], t))
            | None -> None
          else Some ([], ulist)

      let rec forget t =
        match t with
          [] -> []
        | h::t -> (CM.forget h)@((forget t)

    end

Figure 4: Generic definition of the content validation functions.
6. TYPEFUL XML PROGRAMMING IN ML

One of the purposes of validation is to assign a type to an XML element. Programming with validated XML elements is now programming with typed values. Using a statically typed language for such programming allows one to detect type errors, hence expressions for invalid elements, at compile time.

Our generic validation procedure gives types to valid elements, and allows one to construct XML processors in a typeful way. In the following illustrating diagram, let $U$ be the ML type for well-formed elements, and $V$ and $V'$ be the ML types that correspond to specific XML element types.

$$
\begin{array}{c}
U \\
\downarrow \text{validate} \\
V \\
\downarrow f \\
V'
\end{array}
$$

We may say that functions in $U \rightarrow U$ are untyped as they may produce invalid elements. However, functions in $V \rightarrow V'$ are typed as they always output valid elements. Whenever one is programming a function $g : U \rightarrow U$, and expects the output also to be valid, one can do so by programming a function $f : V \rightarrow V'$ so that

$$g = \text{forget} \circ f \circ \text{validate}$$

In Figure 5, we show some ML code fragment to illustrate the approach. The code maps a well-formed tidy bookmark to a well-formed flat bookmark (function tidy2flat_u). Because the the mapping is composed from a typed conversion routine (function tidy2flat_v), it will always output a valid element if the input element is valid. Note that the types for the functions below will be inferred by ML. The functions are annotated with their types in Figure 5 for clarity purpose only.

7. COMBINING GENERICITY WITH POLYMORPHISM

The generic modeling of XML DTDs can be combined with ML type polymorphism for a better result. Indeed, we use both genericity and polymorphism to model XML element type declarations that are accompanied with attribute list declarations. We can extend the previous folder example by requiring an optional subject attribute for each folder element, and a pair of title and url attributes for each record element. The following is a valid XML document with the newly extended DTD.

```
<?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>
```

module TidySys = ... (* See code in Section 2 *)
module FlatSys =
  struct
    module F0 = Plus(F1)
    module F1 = Empty
    module Tag = Tag
  end

module TidyDtd = Mu(TidySys)
module FlatDtd = Mu(FlatSys)

module TidyFolder = TidyDtd.V.T0
module TidyRecord = TidyDtd.V.T1
module FlatFolder = FlatDtd.V.T0
module FlatRecord = FlatDtd.V.T1

let t2f_folder: (FlatFolder.t, FlatRecord.t) TidyFolder.cm ->
   (FlatFolder.t, FlatRecord.t) FlatFolder.cm =
   fun fd -> match fd with
     L ((r, t)...) (* the case of a flat record r followed by a sequence t of flat records or folders *)
   | R ((f, t)...) (* the case of a flat folder f followed by a non-empty sequence t of flat records or folders *)

let t2f_record: (FlatFolder.t, FlatRecord.t) TidyRecord.cm ->
   (FlatFolder.t, FlatRecord.t) FlatRecord.cm =
   fun () -> ()

let flatten_v: (TidyFolder.t, TidyRecord.t) Tag.t ->
   (FlatFolder.t, FlatRecord.t) Tag.t =
   Tag.map (TidyDtd.V.fold (FlatFolder.up $ t2f_folder, FlatRecord.up $ t2f_record))

let flatten_u: TidyDtd.U.t -> FlatDtd.U.t =
   FlatDtd.forget $ flatten_v $ TidyDtd.validate

Figure 5: An example of typeful XML programming.

Note: Type annotations for functions are added for clarity purpose.
and an exception is raised as soon as possible. A content model is ambiguous when its sequence of symbols can be recognized deterministically, by one-symbol lookahead, by the corresponding nondeterministic finite-state machine. For example, the content model \((b, c)\mid (b, d)\) is not 1-ambiguous, because given an initial \(b\), one cannot know which \(b\) in the model is being matched without looking further ahead to see what follows \(b\). However, the equivalent content model \((b, (c|d))\) is 1-ambiguous [2]. We can use the nullable predicate and the first set to check whether the content model as specified by a module expression is 1-ambiguous. The check is performed at module elaboration time so that an ambiguous content model is detected and an exception is raised as soon as possible. A content model may also contain epsilon ambiguity which is allowed by XML but demands additional work during validation. An example of epsilon ambiguity is \((a*|b*)\), when the empty sequence is derivable from both \(a*\) and \(b*\).

Besides element content models (i.e., regular expressions on element type names), an XML element type may use other content specifications. For example, the element type may have EMPTY or ANY specification, or mixed content specification. These specifications impose no additional difficulty in the definition of the generic validation function. The ANY specification means that the sequence of child elements may contain elements of any declared element types, including text, in any order. The mixed content specification allows text data to be interspersed with elements of some prescribed types. One may think of ANY as a special case of mixed content.

One can view text data, which is denoted as \(#PCDATA\) ("Parsed Character Data") in a mixed content specification, as elements enclosed within a pair of implicit \(<text>\) start-tag and \(<\text>\) end-tag. A PCDATA module, similar to the Empty module we already have, can be defined to help inductive definitions of mixed content specifications. For example, for DTDs with 2 element types, one can define an Any module as following by using a 3-ary alternative module Alt3:

```
module Any: FUN = Star(Alt3(P0)(P1)(Pcdata))
```

### 9. EXPERIENCE WITH LARGER DTDs

WML is a markup language for WAP applications. Its DTD consists of 35 element type definitions. We have applied the generic approach to validate WML documents. In order to do so, we need to produce ML modules that include and operate upon 30-ary type constructors (35 element types plus 1 for \(#PCDATA\)). We also need to construct higher-order modules that take in as many as 17 modules as input (one of the element type definitions needs a 17-ary Alt module). Our experience has been quite satisfactory: Our code is compiled without problem with Objective Caml, but the compilation time is not negligible (about 1 min. at a desktop Sparc workstation). The validation time is negligible however, at least for the smallish examples we have tried (around 100 elements). We are working on both larger DTDs and documents, and are collecting more performance data.

The size of the ML source code is quite large, however. Take the following ML module expression as an example.

```
module F10 = Seq10(P0)(P1)(P2)(P3)(P4)
            (P5)(P6)(P7)(P8)(P9)
```

One need a 10-ary module Seq10 to construct the required content model, which specifies a sequence of 10 elements, each of a different element type. Code for module Seq10 looks like the following:

```
module Seq10 = functor (F0: FUN) ->
            functor (F1: FUN) -> ...
            functor (F9: FUN) ->
struct
  type ('x0, 'x1, ..., 'x35) t
    = ('x0, 'x1, ..., 'x35) F0.t
    * ('x0, 'x1, ..., 'x35) F1.t
    ...
    * ('x0, 'x1, ..., 'x35) F9.t
...
end
```
It is clear from the above that, for a DTD with $n$ element types, the source for module Seq$_n$ will have code size $O(mn)$. At the worst case, for a DTD of length $n$, our code will need $O(n)$ unique type variables, will contain type sharing constraints of length $O(n^2)$, and will have an overall code size of $O(n^3)$. The source code of all the necessary ML modules for the 35-element WML DTD has a size of about 0.5 MB. When compiled, it produces a binary of size 175 KB (*.cmo file in Objective Caml), and an interface of size 2.3 MB (*.cmi file in Objective Caml). ML code for the WAP examples is accessible at the following URL:

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

One can do a connected component analysis on the DTD so that the set of element types are partitioned into disjoint subsets where there is no type-dependency between the subsets. A subset with $k$ element types need only use $k$-ary type constructors, and the overall code size for the modules used for the subset can be reduced.

10. RELATED WORK AND CONCLUSION

In Section 1, we have introduced previous work that uses existing or new functional languages to model and program with XML DTDs. There is a wealth of research and system work that is related to XML content modeling but is not necessarily from the perspective of (functional) programming languages. We list just a few here.

Brüggemann-Klein and Wood addressed the problem of ambiguous XML (and SGML) content models, based on theory of regular languages and finite automata [7, 8]. In particular, they showed that linear time suffices to decide whether a content model is ambiguous. It is shown that regular expressions in both “star normal form” and “epsilon normal form” are always unambiguous [9]. The Glushkov automaton that corresponds to a regular expression is used for checking ambiguity and, if not unambiguous, for validation as well. Murata has proposed a data model for XML document transformation that is based on forest-regular language theory [15, 16]. His model is a lightweight alternative to XML Schema and provides a framework for schema transformation. There is also work on type modelling for document transformation in a structured editing systems using data types [5]. However, none of the above work has used specific programming language as a modeling language.

XML Schema is a maturing specification language for XML content that is being developed at World Wide Web Consortium [4]. XML Schema is more expressive than DTD and the specification language itself uses XML syntax. The key difference between XML Schema and DTD seems to be XML Schema's ability to derive new types by extending or restricting the content models of existing types. XML Schema also provides a “substitution groups” mechanism to allow elements to be substituted for other elements. We are investigating whether ML-like module languages are expressive enough to model these mechanisms.

Backhouse, Jansson, and Jeuring, and Meertens have written a detailed introduction to generic programming [6]. See also the introduction to fold/unfold by Meijer, Fokkinga, and Paterson [13], as well as work on using fold/unfold for structuring and reasoning about program semantics by Hutton [12]. Our extension of simple fold to simultaneous fold seems new. Most work about generic programming in the functional programming research community seems to rely on the mechanism of type class to derive type-specific instances of generic functions. The language of choice is often Haskell. We have shown in this paper that the parametric module mechanism in ML-like languages is suitable for generic programming as well. In fact, we think that parametric modules allow one to take finer control on the inductive derivations of generic values. More powerful module systems have been developed to allow mutually recursive modules, as well as modules that depend on values and types (see, e.g., Russo [17]). However, we showed here that the lack of recursive modules need not be a problem as long as the mutual dependency between the modules is only about interdependent type definitions.

Viewed in the above context, our work can be thought to use the ML module facility to generate a deterministic automata that is specialized for the validation of elements for a specific DTD. Validation automata also gives types to the elements (and its parts). In additional, the construction of the validation automata is entirely generic and can be automated. Our work also serves as a usage case of ML parametric modules, and can be used to stress test current ML implementations. It is a delight to see our contrived code of 36-ary type constructors and 17-ary higher-order modules is compiled and executed with no problem under Objective Caml.

11. REFERENCES


