Overview

- XML Schema Languages
  - XML DTD
  - XML Schema
  - XDR
  - SOX
  - Schematron
  - DSD
- Haskell and XML- HaXML
- XML Schema - Haskell Data Binding- UUXML

Why we need XML Schema Languages

- Increase of the amount of data in XML format
- XML is the core technology of modern data exchange
- XML document is a tree-based data structure, not necessarily structured according to a type declaration such as a schema
- XML Schema languages describe XML data structures and constraints

Schema Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Syntax in XML	No	Yes	Yes	Yes	Yes	Yes
Namespace	No	Yes	Yes	Yes	Yes	No
Include	No	Yes	No	Yes	No	Yes
Import	No	Yes	No	Yes	No	No

Datatype Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Built-in Types	10	37	33	17	0	0
User-defined Types	No	Yes	No	Yes	No	Yes
Domain Constraint	No	Yes	No	Partial	Yes	Yes
Null	No	Yes	No	No	No	No

Attribute Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Default Value	Yes	Yes	Yes	Yes	No	Yes
Choice	No	No	No	No	Yes	Yes
Optional vs Required	Yes	Yes	Yes	Yes	Yes	Yes
Domain Constraint	Partial	Yes	Partial	Partial	Yes	Yes
Conditional Definition	No	No	No	No	Yes	Yes

Element Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Default Value	No	Partial	No	No	No	Yes
Content Model	Yes	Yes	Yes	Partial	Yes	Yes
Ordered Sequence	Yes	Yes	Yes	Yes	Yes	Yes
Unordered Sequence	No	Yes	Yes	No	Yes	Yes
Choice	Yes	Yes	Yes	Yes	Yes	Yes
Min & Max ocurrence	Partial	Yes	Yes	Yes	Yes	Partial
Open Model	No	No	Yes	No	Yes	No
Conditional Definition	No	No	No	No	Yes	Yes

Inheritance Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Simple Type by Extension	No	No	No	No	No	No
Simple Type by Restriction	No	Yes	No	Yes	No	No
Complex Type by Extension	No	Yes	No	Yes	No	No
Complex Type by Restriction	No	Yes	No	No	No	No

Being unique or key Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Uniqueness for attribute	No	Yes	Yes	Yes	Yes	Yes
Uniqueness for non-attribute	No	Yes	Partial	No	Yes	No
Key for attribute	No	Yes	No	No	Yes	No
Key for non-attribute	No	Yes	No	No	Yes	No
Foreign Key for attribute	Partial	Yes	Partial	Partial	Yes	Yes
Foreign Key for non-attribute	No	Yes	No	No	No	Yes

Miscellaneous Features

	DTD 1.0	XML Schema 1.0	XDR 1.0	SOX 2.0	Schematron 1.4	DSD 1.0
Dynamic con-straint	No	No	No	No	Yes	No
Version	No	No	No	No	No	Yes
Documentation	No	Yes	No	Yes	No	Yes
Embedded HTML	No	Yes	No	Yes	Partial	Yes
Self-describability	No	Partial	No	No	Partial	Yes

DTD Example

<!ATTLIST list type (bullets|ordered) 'ordered'>
<!ATTLIST form method CDATA #FIXED 'POST'>

Attribute: Default Value

<!ATTLIST spec RGB (red|green|blue)>

Attribute: Domain Constraint

---

empty : <!ELEMENT o EMPTY>
text : <!ELEMENT p (#PCDATA) >
element : <!ELEMENT q (x?|y*|z+) >
mixed : <!ELEMENT r (#PCDATA|x)* >

Element: Content Model

XML Schema Example

<schema xmlns:z='URI' ...> <element name='book'> <complexType> <element name='title' type='string'> <element name='address' type='z:address'> </complexType> </element> </schema>	Schema: Namespace
schema : <element name='fullname' nullable='true'/> instance: <fullname xsi:null='true'></fullname>	Datatype: Null
<attribute name='nm' use='default' value='John Doe'/>	Attribute: Default Value
<element name='fullname' type='string' default='John Doe'/>	Element: Default Value
<simpleType name='zipcode' base='string'> <pattern value='[0-9]5(-[0-9]4)?'/> </simpleType>	Inheritance: Simple type by restriction

XML Schema Example

<complexType name='ItemType'> <element name='item' minOccurs='0'> </complexType> <complexType name='ResItemType' base='ItemType' derivedBy='restriction'> <element name='item' minOccurs='1'> </complexType> <element name='E' type='ResItemType'>	Inheritance: Complex type by restriction
<unique> <selector>person/addr</selector> <field>phone</field> </unique>	Being unique or key: Uniqueness for non-attribute
<key name='ekey'> <selector>employee</selector> <field>dept/code</field> <field>@name</field> </key>	Being unique or key: Key for non-attribute
<keyref refer='ekey'> <selector>project</selector> <field>empdept</field> <field>@ename</field> </keyref>	Being unique or key: Foreign Key for non-attribute

XDR Example

<ElementType name='title' dt:type='string'/> <ElementType name='book' xmlns:z='URI'> <element type='title'><element type='z:address'> </ElementType>	Schema: Names-pace
<AttributeType name='nm' dt:type='string'/> <attribute type='fullname' default='John Doe'/>	Attribute: Default Value
<AttributeType name='RGB' dt:type='enumeration' dt:values='red green blue'/>	Attribute: Domain constraint

SOX Example

<namespace prefix='z' namespace='URI'/> <elementtype name='book'> <model> <element type='title'> <element prefix='z' type='address'> </model> </elementtype>	Schema: Namespace
<attrdef name='nm' datatype='string'> <default>John Doe</default> </attrdef>	Attribute: Default
<elementtype name='person'> <model><sequence> <element name='fn'/><element name='ln'/> </sequence></model> </elementtype>	Element: Ordered Sequence
<datatype name='RGB'> <enumeration datatype='color'> <option>Red</option> <option>Green</option> <option>Blue</option> </enumeration> </datatype>	Inheritance: Simple type by restriction
<elementtype name='new-person'> <extends type='person'> <append> <element name='address' type='addr'/> </append> <attdef name='email' datatype='string'> </extends> </elementtype>	Inheritance: Complex type by extension

Schematron Example

<rule context='E'> <assert test='floor(.) = number(.)'> E can have only integer value.</assert> </rule>	Datatype: Domain Constraint
<rule context='person'> <assert test='@fn or @gn'>Or semantics</assert> <assert test='count(attribute::*) = 1'> Only one attribute</assert> </rule>	Attribute: Choice Among Attributes
<rule context='E'> <report test='(@one) or not(@one and @two)'> E cannot have attribute 'one' alone.</report> </rule>	Attribute: Conditional Definition
<rule context='person'> <assert test='(*[position()=1] = fn) and (*[position()=2] = ln)'> fn must be followed by ln.</assert> </rule>	Element: Ordered Sequence

Schematron Example

<rule context='person'> <assert test='count(name|address) = count(*)'> There is an extra element.</assert> </rule>	Element: Open Model
<rule context='E'> <report test='not(parent::form) and input'> Element input cannot appear.</report> </rule>	Element: Conditional Definition
<rule context='person/addr'> <assert test='count(phone) = 1'> phone is not unique.</assert> </rule>	Being unique or key: Uniqueness for non-attribute
<rule context='person'> <assert test='@ssn and count(@ssn) = 1'> Is ssn unique?</assert> <assert test='string-length(@ssn) > 0'> Is ssn not empty?</assert> </rule>	Being unique or key: Key for attribute
<rule context = 'employee[@dno]'> <assert test='(name(id(@dno)) = 'dept')'> Error occurred.</assert> </rule>	Being unique or key: Foreign key for attribute

DSD Example

<StringTypeDef ID=’zipcode’> <Sequence> <Repeat value=’5’> <CharSet Value=’0123456789’> </Repeat> <Optional> <String Value=’-’> <Repeat value=’4’> <CharSet Value=’0123456789’> </Repeat> </Optional> </Sequence> </StringTypeDef>	Datatype: User-Defined Types
<Default> <Context><Element Name=’employee’> <Attribute Name=’gender’ Value=’M’/> </Element></Context> <DefaultAttribute Name=’nm’ Value=’John Doe’/> </Default>	Attribute: Default Value
<ElementDef ID=’person’> <AttributeDecl Name=’fn’ IDType=’ID’/> <AttributeDecl Name=’gn’ IDType=’ID’/> <OneOf> <Attribute Name=’fn’/><Attribute Name=’gn’/> </OneOf> </ElementDef>	Attribute: Choice among attributes
<ElementDef ID=’student’> <If><Attribute Name=’TA’ Value=’yes’> <Then><Optional> <AttributeDecl Name=’salary’/> </Optional></Then> </If> </ElementDef>	Attribute: Conditional Definition

DSD Example

<Default> <Context><Element Name='address'/></Context> <DefaultContent> <city>Los Angeles</city><state>CA</state> </DefaultContent> </Default>	Element: Default value
<ElementDef ID='person'> <Sequence> <Element Name='fn'/><Element Name='ln'/> </Sequence> </Element-Def>	Element: Ordered Se-quence
<ElementDef ID='person'> <OneOf> <Element Name='fn'/><Element Name='gn'/> </OneOf> </ElementDef>	Element: Choice among elements
<ConstraintDef ID='book-constraints'/> <ConstraintDef RenewID='book-constraints'> <Constraint CurrIDRef='book-constraints'/> ... modification ... </ConstraintDef>	Being unique or key: Foreign key for non-attribute

Conclusion

- DTD is the easiest but has the weakest expressive power
- XDR and SOX support schema datatype but fail to support constraint specifi cation to express the semantics of schema
- XML Shema, Schematron and DSD have the strongest expressive power
- No single Schema language suffi ces the needs from a database point of view

HaXML

- XML Document Structure
  - All markup has an explicit end marker
  - Every document is *well-formed*, have a nesting structure sintactically clear
  - Parts of the document can be selected, re-arranged, transformed by structure
  - The start tag may contain attributes, in the form name/value
- Representing XML in Haskell
  - Generic combinators
  - Translation of DTDs to Types

Documents and Transformations

- **Data modelling**

  data Element = Elem Name [Attribute] [Content]

  data Content = CElem Element | CText String

- **The filter type**

  type CFilter = Content -> [Content]

- **Program wrapper**

  processXMLWith :: CFilter -> IO ()

- **Basic filters**
  - Predicate Filters
  - Selection Filters
  - Construction Filters

Basic Content Filters

- Predicates

```
none,                          zero/failure
keep,                          identity/success
elm,                           tagged element ?
txt,                           plain text ?
::CFilter
tag                            named element ?
attr                           element has attribute ?
:: String -> CFilter
attrval                        element has attribute/value ?
:: (String,String) -> CFilter
```

- Selection

```
children                children of element
CFilter
showAttr,               value of attribute
(?)                     synonym for showAttr
:: String -> CFilter
```

Basic Content Filters

- Construction

literal,	build plain text
(!)	synonym for literal
:: String -> CFilter	
mkElem	build element
:: String -> [CFilter] -> CFilter	
mkElemAttrs	build element with atrributes
:: String -> [(String,CFilter)] -> [CFilter] -> CFilter	
replaceTag	replace element's tag
:: String -> CFilter	
replaceAttrs	replace element's attributes
:. [(String,CFilter)] -> CFilter	

Combinators

- combinators are higher-order operators assembling functions into more powerful combinations
- uniform combinators make possible to compose any function with any other
- basic fi lters can be combined into more interesting and complex fi lters

Filter Combinators

o,	Irish composition
(|||),	append results
with	guard
without	negative guard
(/>),	interior search
(</),	exterior search
(|>|)	directed search
:: CFilter -> CFilter -> CFilter	
cat	concatenate results
:: [CFilter] -> CFilter	
et	disjoint union
:: (String -> CFilter) -> CFilter -> CFilter	
(?>)	if-then-else choice
:: CFilter -> ThenElse CFilter -> CFilter	

Filter Combinators

chip,	"in-place" application to children
deep,	recursive search (*topmost*)
deepest,	recursive search (*deepmost*)
multi,	recursive search (*all*)
foldXml	recursive application

:: CFilter -> CFilter

Translation of DTDs to Types

- When the DTD for a document is available, the meaning it defines for markup tags can be used to powerful effect
- Confirm semantic *validity*
- DTD defines a grammar for document content: vocabulary for markup tags, and the allowed content and attributes for each tag.
- With documents validated with DTD, we can write *valid processing scripts*, scripts that produce a valid document as output, given a valid document as input.
- Define a correspondence between the DTD of a document and a set of algebraic types in Haskell, and the consequent correspondence between the document's content and a structured Haskell value.
- By writing document processing scripts to manipulate the typed Haskell value, the script validation problem is just an instance of normal Haskell type inference (?).

DTD translations

- Elements
  - Each element declaration is translated to a new datatype declaration
  - Sequence is translated to product types (single-constructor values)
  - Choice is translated to co-product types (multi-constructor values)
  - Optionality is simple translated to **Maybe** type
- Attributes
  - Attribute lists are unordered and accessed by name, so Haskell named-fields are a good representation
  - Optionality is again translated as **Maybe** type
  - Attribute values that are constrained to a particular set of values can be modelled are translated as a new enumeration type with the permitted strings.

Pros and Cons of the two schemes

Combinators

Ease of extension and variation

Computational power

Abstraction, generality and reuse

Laws for reasoning about scripts

Implementation for free

Distance from target language

Living in an unfamiliar world

Type-based translation

Validity

Direct programing style

High startup cost

Incomplete type model

XML Schema-Haskell Data Binding

- A data binding has signifi cant advantages over the SAX or DOM aproach
  - Parsing comes for free and can be optimized for a specifi c schema
  - It is easier to implement, test and maintain software in the target language
- UUXML is a translation of XML documents into Haskell, and more specifi cally a translation tailored to permit writing programs in Generic Haskell
- The document conform to the type system described in the W3C XML Schema, and the translation preserves typing

Schema Types

g::=**group**

	ϵ	empty sequence
|	g,g	sequence
|		empty choice
|	g | g	choice
|	g & g	interleaving
|	g{m,n}	repetition
|	**mix**(g)	mixed content
|	x	component name
|	g | g	choice

m ::= <natural> **minimum**

x ::=

	@a	attribute name
|	e	element name
|	t	type name
|	**anyType**	
|	**anyElem**	
|	**anySimpleType**	
|	p	primitive

n ::= **maximum**

	m	bounded
|	∞	unbounded

Translation

$$
\begin{aligned}
[\epsilon]_G &= () \\
[g_1, g_2]_G &= ([g_1]_G, [g_1]_G) \\
[g^*]_G &= [[g_1]_G] \\
[g?]_G &= \textbf{Maybe}\ [g]_G \\
[]_G &= \textbf{Void} \\
[g_1|g_2]_G &= \textbf{Either}\ [g_1]_G [g_2]_G \\
[g^+]_G &= ([g]_G, [g^*]_G)
\end{aligned}
$$

Translation of the XML Schema feature

- **Primitives** are translated to the correnponding Haskell types
- **User-Defined types** are translated, using products to model sequences and co-products (sums) to model choices
- **Mixed Content**
- **Interleaving** is modeled in essentially the same way as sequencing, with a different abstract datatype. Just parsing is different, must considerer permutations.
- **Repetition** are modeled using a datatype Rep $[g]_G[m,n]_B u$ and a set of datatypes modeling bounds

  $[0,0]_B = ZZ$

  $[0,\infty]_B = ZI$

  $[0,m+1] = ZS[0,m]_B$

  $[m+1,n+1]_B = SS[m,n]_B$