XQuery: Formal Semantics and Types

References

Item types (elements, in particular) and sequence types

- Element types: element name + content type (a sequence type)
- Sequence type: **name** and structure
- One is defined in terms of the other
- **define element** paper of type paperType
- **define type** paperType {
  element title,
  element author +
}
- **define element** title of type xs:string
- **define element** author of type xs:string

Type: sequence types

- Regular expressions based on item types
- Type ::= ItemType
  | Type (? | * | +) 
  | Type , Type 
  | Type | Type 
  | ()
- ItemType ::= AtomicTypeName | ElementType
- ElementType ::= **element** ElementName? (of type TypeName)?
Defining element types and sequence types

- **define element** `ElemName of type TypeName`
- **define type** `TypeName TypeDerivation`
- `TypeDerivation ::=`
  - `restricts` `AtomicTypeName`
  - `restricts` `TypeName { Type }`
  - `extends` `TypeName { Type }

- **define element** `paper of type paperType`
- **define type** `paperType [restrict Anytype] { element title, element author + }

- Abbreviated form:
  - **define element** `paper restrict AnyType { element title, element author + }

Element validation

- The environment associates element names to a sequence-type name and a sequence-type structure
- Matching / validation:
  - The element name must be equal to the element-type name (unless it is *)
  - The element content is associated to a sequence-type name:
    - Based on the element name
    - But the structure must correspond
Named and anonymous sequence types

• **element** paper has ‘paperType’ as its sequence type:
  – define element paper of type paperType
  – define type paperType {
    element title,
    element author +
  }

• **element** paper has an anonymous sequence type:
  – define element paper restrict AnyType {
    element title,
    element author +
  }

Anonymous content types

• Anonymous types have a hidden name that is automatically generated

• The two ‘**element** a’ element types have different sequence types:
  • define element a {xs:Integer}
  • define element integerContainer {
    element a {xs:Integer},
    element b {xs:Integer}
  }
Representing XDM values

• `<fact>The cat weighs <weight units="lbs">12</weight> pounds.</fact>`

• `element fact of type xs:untyped {
  text { "The cat weighs " },
  element weight of type xs:untyped {
    attribute units of type xs:untypedAtomic {
      "lbs" of type xs:untypedAtomic
    },
    text { "12" }
  },
  text { " pounds." }
}`

Matching

• The problem: `element weight of type weightType {
  text { "42" }
}` can be passed to a function that expects an ‘elementTypeX’ parameter?

• Matching: look for the meaning ‘element k {T}’ of ‘elementTypeX’ and check whether ‘weight’ respects k, and whether the value `text{"42"}` respects T

• Validation: is the process of adding ‘of type weightType’ to an untyped element
Matching

**ElementType** yields **BaseElemName** of type **BaseTName**

**BaseTName** resolves to **Type**

**AttributeName** substitutes for **BaseElemName**

**TypeName** derives from **BaseTName**

Value matches **Type**

<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>element <strong>AttributeName</strong> of type <strong>TypeName</strong> { Value } matches <strong>ElementType</strong></td>
</tr>
</tbody>
</table>

**ElementType** yields **BaseElementName** of type **BaseTypeName**

- **yields** expand ‘elementType’ to ‘name– type’
- If **ElementType** is an **AttributeName** with global definition: **yields** returns its definition
- Otherwise:
  - element **AttributeName** OfType **yields** **AttributeName** OfType
  - element OfType **yields** * OfType
  - element **yields** * of type xs:anyType
**TypeName resolves to Type**

- **resolves to**: expands TypeName to a sequence type by concatenating the extensions
- **define type** TypeName restricts _ {Type} => TypeName resolves to Type
- **define type** TypeName extends BaseName {Type} BaseName resolves to BaseType => TypeName resolves to BaseType, Type

**ElementName substitutes for BaseElementName**

- elementName substitutes for elementName
- elementName substitutes for *
TypeName **derives from** BaseTypeName

• Assume this definition:
  – **define type** TypeName **restricts/extends** BaseTypeName {Type}

  then:
  – TypeName **derives from** BaseTypeName

• Closed by reflexivity and transitivity

---

**Matching**

**ElementType yields** BaseElemName **of type** BaseTName

BaseTName **resolves to** TypeExpression

**ElementName substitutes for** BaseElemName

TypeName **derives from** BaseTName

**Value matches** TypeExpression

________________________________________________________________________

element **ElementName** **of type** TypeName {Value} **matches** ElementType
Validation

• Before validation:
  – element weight of type xs:untyped {
    attribute xsi:type of type xs:untypedAtomic {
      "xs:integer" of type xs:untypedAtomic
    },
    text { "42" }
  }
• After validation:
  – element weight of type xs:integer {
    attribute xsi:type of type xs:QName {
      "xs:integer" of type xs:QName
    },
    42 of type xs:integer
  }

Simple types: list types

• Simple types give a type to sequences of atomic values
• List type:
  – define element sizes of type sizesType;
  – define type sizesType {list of sizeType};
  – define type sizeType restricts xs:integer;
Use of a list type

- define element sizes of type sizesType;
- element sizes of type xs:untyped
  { text { "1 2 3" } }
- After validation:
  - element sizes of type sizesType {
    1 of type sizeType,
    2 of type sizeType,
    3 of type sizeType
  }

Normalization

- for $i$ in (1, 2),
  $j$ in (3, 4)
  return element pair { ($i$, $j$) }
- Is normalized to:
  - for $i$ in (1, 2) return
    for $j$ in (3, 4) return
    element pair { ($i$, $j$) }