XRPC: efficient distributed query processing on heterogeneous XQuery engines

Zhang, Y.

Citation for published version (APA):
Zhang, Y. (2010). XRPC: efficient distributed query processing on heterogeneous XQuery engines

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
XML Schema Definition of the XRPC SOAP Messages

```xml
<xsd:schema version="0.1" xml:lang="EN" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xrpc="http://monetdb.cwi.nl/XQuery" elementFormDefault="qualified"
    targetNamespace="http://monetdb.cwi.nl/XQuery">

    <xsd:element name="atomic-value" type="xsd:anySimpleType"/>
    <xsd:element name="attribute"/>
    <xsd:element name="comment"/>
    <xsd:element name="document"/>
    <xsd:element name="element"/>
    <xsd:element name="processing-instruction"/>
    <xsd:element name="text"/>
    <xsd:element name="udf-element" type="xrpc:udfElemType"/>
    <xsd:element name="request" type="xrpc:requestType"/>

    <xsd:complexType name="qidType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="requestType">
        <xsd:attribute name="host" type="xs:anyURI"/>
        <xsd:attribute name="timestamp" type="xs:dateTime"/>
        <xsd:attribute name="timeout" type="xs:double"/>
        <xsd:attribute name="module" type="xs:string"/>
        <xsd:attribute name="method" type="xs:string"/>
        <xsd:attribute name="location" type="xs:anyURI"/>
        <xsd:attribute name="arity" type="xs:integer"/>
        <xsd:attribute name="iter-count" type="xs:integer"/>
        <xsd:attribute name="updCall" type="xs:string"/>
        <xsd:attribute name="tag" type="xs:string"/>
        <xsd:attribute name="caller" type="xs:string"/>
    </xsd:complexType>

    <xsd:complexType name="callType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="callResponseType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="callSequenceType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="callQueryIDType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="queryResponseType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

    <xsd:complexType name="sequenceType">
        <xsd:attribute ref="xrpc:timestamp" use="required"/>
        <xsd:attribute ref="xrpc:host" use="required"/>
        <xsd:attribute ref="xrpc:timeout" use="required"/>
    </xsd:complexType>

</xsd:schema>
```
<xs:complexType name="udfElemType">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="sequenceType">
  <xs:sequence minOccurs="0" maxOccurs="unbounded">
    <xs:choice>
      <xs:element ref="xrpc:atomic-value"/>
      <xs:element ref="xrpc:attribute"/>
      <xs:element ref="xrpc:comment"/>
      <xs:element ref="xrpc:document"/>
      <xs:element ref="xrpc:element"/>
      <xs:element ref="xrpc:processing-instruction"/>
      <xs:element ref="xrpc:text"/>
      <xs:element ref="xrpc:udf-element"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="callType">
  <xs:sequence>
    <xs:element ref="xrpc:sequence" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="requestType">
  <xs:sequence>
    <xs:element ref="xrpc:queryID" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="xrpc:call" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="responseType">
  <xs:sequence>
    <xs:element ref="xrpc:queryID" minOccurs="0" maxOccurs="1"/>
    <xs:element ref="xrpc:sequence" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:schema>
</xs:schema>
XQuery Implementation Defined and Implementation Dependent Features

In this chapter, we give a complete list of implementation defined and implementation dependent features defined by the XQuery specifications, including [XQuery 1.0 and XPath 2.0 Data Model] [71], [XQuery 1.0: An XML Query Language] [38], [XQuery 1.0 and XPath 2.0 Functions and Operators] [124], [XSLT 2.0 and XQuery 1.0 Serialization] [39], and [XQuery Update Facility 1.0] [58].

B.1 XQuery 1.0 and XPath 2.0 Data Model

Implementation Defined Features

• Support for additional user-defined or implementation defined types.
• Some typed values in the data model are undefined. Attempting to access an undefined property is always an error. Behavior in these cases is implementation defined and the host language is responsible for determining the result.
• The internal structure of the values of the unparsed-entities property.
• When mapping a Document Node to a document information item, the declaration base URI property of the unparsed entities property.
• When constructing the children property for an Element Node from a PSVI, the relative order of Processing Instruction and Comment Nodes must be preserved, but the position of the Text Node, if it is present, among them is implementation defined.

Implementation Dependent Features

• The relative order of Namespace Nodes nodes is stable but implementation dependent.
• The relative order of Attribute Nodes nodes is stable but implementation dependent.
• The relative order of nodes in distinct trees is stable but implementation dependent.
• The names of anonymous types.
• The prefix associated with type names.
• The representation of the set of prefix/URI pairs returned by the \texttt{dm:namespace-bindings} accessor.

• The representation of namespaces, i.e. whether or not they are represented as nodes.

• When constructing the \textit{children} property for an Element Node from a PSVI, where a fixed or default value for an element is defined in the schema, and the element takes this default value, a text node will be created to contain the value, even though there are no character information items representing the value in the PSVI. The position of this text node relative to any comment or processing instruction children is implementation dependent.

\section*{B.2 XQuery 1.0: An XML Query Language}

\textbf{Implementation Defined Features}

• The \textit{version of Unicode} that is used to construct expressions.

• The \textit{statically-known collations}.

• The \textit{implicit timezone}.

• The circumstances in which warnings are raised, and the ways in which warnings are handled.

• The method by which errors are reported to the external processing environment.

• Whether the implementation is based on the rules of [XML 1.0][49] and [XML Names][47] or the rules of [XML 1.1][50] and [XML Names 1.1][48] is implementation defined. One of these sets of rules must be applied consistently by all aspects of the implementation.

• Any components of the static context or dynamic context that are overwritten or augmented by the implementation.

• Which of the optional axes are supported by the implementation, if the Full-Axis Feature is not supported.

• The default handling of empty sequences returned by an ordering key (sortspec) in an \textit{order by clause} (empty least or empty greatest).

• The names and semantics of any \textit{extension expressions} (pragmas) recognised by the implementation.

• The names and semantics of any option declarations recognised by the implementation.

• Protocols (if any) by which parameters can be passed to an external function, and the result of the function can returned to the invoking query.

• The process by which the specific modules to be imported by a module import are identified, if the Module Feature is supported (includes processing of location hints, if any.).

• Each \textit{module import} names a target namespace and imports an implementation defined set of modules that share this target namespace.

• In a module import, the \texttt{URILiteral}s that follow the \texttt{at} keyword are optional location hints, and can be interpreted or disregarded in an implementation defined way.

• Any static typing extensions supported by the implementation, if the Static Typing Feature is supported.
• The means by which serialisation is invoked, if the Serialization Feature is supported.

• The default values for the byte-order-mark, media-type, normalization-form, encoding, omit-xml-declaration, standalone and version parameters, if the Serialization Feature is supported.

• The result of an unsuccessful call to an external function (for example, if the function implementation cannot be found or does not return a value of the declared type).

• The following limits on ranges of values is implementation defined:
  ◦ For the xs:decimal type, the maximum number of decimal digits (totalDigits facet) (must be at least 18).
  ◦ For the types xs:date, xs:time, xs:dateTime, xs:gYear, and xs:gYearMonth: the maximum value of the year component and the maximum number of fractional second digits (must be at least 3).
  ◦ For the xs:duration type: the maximum absolute values of the years, months, days, hours, minutes, and seconds components.
  ◦ For the xs:yearMonthDuration type: the maximum absolute value, expressed as an integer number of months.
  ◦ For the xs:dayTimeDuration type: the maximum absolute value, expressed as a decimal number of seconds.
  ◦ For the types xs:hexBinary, xs:base64Binary, xs:QName, xs:anyURI, xs:NOTATION, xs:string and types derived from them: limitations (if any) imposed by the implementation on lengths of values.

• The in-scope variables may be augmented by implementation defined variables.

Implementation Dependent Features

• If an implementation does not support the Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression, if evaluated, will necessarily raise a type error at run time, the implementation may raise that error during the static analysis phase. The choice of whether to raise such an error at analysis time is implementation dependent.

• Each schema type definition is identified either by an expanded QName (for a named type) or by an implementation dependent type identifier (for an anonymous type).

• Each element declaration is identified either by an expanded QName (for a top-level element declaration) or by an implementation dependent element identifier (for a local element declaration).

• Each attribute declaration is identified either by an expanded QName (for a top-level attribute declaration) or by an implementation dependent attribute identifier (for a local attribute declaration).

• The function implementation for a built-in function or external function,

• The current dateTime represents an implementation dependent point in time during the processing of a query, and includes an explicit timezone.
- During the static analysis phase, if the Static Typing Feature is not supported, the static types that are assigned to expressions are implementation dependent.
- In addition to the errors defined in this specification, an implementation may raise a dynamic error for a reason beyond the scope of this specification. For example, limitations may exist on the maximum numbers or sizes of various objects. Any such limitations, and the consequences of exceeding them, are implementation dependent.
- The fn:collection() function with zero arguments returns the default collection, an implementation dependent sequence of nodes.
- When an unknown schema type is encountered during the process of SequenceType matching, an implementation is allowed (but is not required) to provide an implementation dependent mechanism for determining whether the unknown schema type is derived from the expected schema type.
- When evaluating the argument expressions of a function call, the order of argument evaluation is implementation dependent and a function need not evaluate an argument if the function can evaluate its body without evaluating that argument.
- The order of the returned sequence of a path expression, if ordering mode is not ordered.
- If ordering mode is not ordered, the resulting sequence of the operators union, intersect, "|" and except is returned in implementation dependent order.
- The order, in which the operands of an arithmetic expression (i.e., UnaryExpr, ValueExpr, AdditiveExpr and MultiplicativeExpr) are evaluated.
- The order, in which the operands of a value comparison are evaluated.
- The order, in which the operands of a node comparison are evaluated.
- The order, in which the operands of a logical expression are evaluated.
- For each namespace used in the name of a constructed element or in the names of its attributes, a namespace binding must exist. If a namespace binding does not already exist for one of these namespaces, a new namespace binding is created for it. If the name of the node includes a prefix, that prefix is used in the namespace binding; if the name has no prefix, then a binding is created for the empty prefix. If this would result in a conflict, because it would require two different bindings of the same prefix, then the prefix used in the node name is changed to an arbitrary implementation dependent prefix that does not cause such a conflict, and a namespace binding is created for this new prefix.
- In a for clause, if the ordering mode is not ordered, the ordering of the variable bindings is implementation dependent.
- In the order by and return clauses, when two orderspec values are compared to determine their relative position in the ordering sequence, the order of two tuples T1 and T2 in the tuple stream is implementation dependent, if neither of the first pair of values encountered during the evaluation is “greater-than” the other, and if stable is not specified.
- The order, in which test expressions of a quantified expression (i.e., QuantifiedExpr) are evaluated for the various binding tuples, is implementation dependent.
- When evaluating a cast expression (i.e., CastExpr), an implementation may determine that one type is derived by restriction from another type either by examining the in-scope schema
definitions or by using an alternative, implementation dependent mechanism such as a data dictionary.

- The handling of an encoding declaration.

- If a version declaration is present, no XQuery Comment may occur before the end of the version declaration. If such a Comment is present, the result is implementation dependent.

- In a schema import, the URILiterals that follow the at keyword are optional location hints, and can be interpreted or disregarded in an implementation dependent way.

### B.3 XQuery 1.0 and XPath 2.0 Functions and Operators

#### Implementation Defined Features

- The destination of the trace output.

- For xs:integer operations, implementations that support limited-precision integer operations must either raise an error [err:FOAR0002] or provide an implementation defined mechanism that allows users to choose between raising an error and returning a result that is modulo the largest representable integer value.

- For xs:decimal values the number of digits of precision returned by the numeric operators is implementation defined.

- If the number of digits in the result of a numeric operation exceeds the number of digits that the implementation supports, the result is truncated or rounded in an implementation defined manner.

- It is implementation defined which version of Unicode is supported by the features defined in this specification, but it is recommended that the most recent version of Unicode be used.

- For fn:normalize-unicode(), conforming implementations must support normalisation form “NFC” and may support normalisation forms “NFD”, “NFKC”, “NFKD”, “FULLY-NORMALIZED”. They may also support other normalisation forms with implementation defined semantics.

- The ability to decompose strings into collation units suitable for substring matching is an implementation defined property of a collation.

- All minimally conforming processors must support year values with a minimum of 4 digits (i.e., YYYY) and a minimum fractional second precision of 1 millisecond or three digits (i.e., s.sss). However, conforming processors may set larger implementation defined limits on the maximum number of digits they support in these two situations.

- The result of casting a string to xs:decimal, when the resulting value is not too large or too small but nevertheless has too many decimal digits to be accurately represented, is implementation defined.

- Various aspects of the processing provided by fn:doc() are implementation defined. Implementations may provide external configuration options that allow any aspect of the processing to be controlled by the user.

- The manner in which implementations provide options to weaken the stable characteristic of fn:collection() and fn:doc() are implementation defined.
Implementation Dependent Features

- For `fn:error()`, the method by which the `xs:anyURI` or `xs:QName` is returned to the external processing environment is implementation dependent.

- For `fn:error()`, if an invocation provides `$description` and `$error-object`, then these values may also be returned to the external processing environment. The method by which these values are provided to the external environment is implementation dependent.

- The format of the trace output is implementation dependent.

- The ordering of output from invocations of the `fn:trace()` function.

- For `fn:distinct-values()`, the order in which the distinct values are returned.

- For `fn:distinct-values()`, which value of a set of values that compare equal is returned.

- The function `fn:unordered()` returns the items of its parameter `$sourceSeq` in an implementation dependent order.

- The function `fn:max()` selects an item from the input sequence `$arg` whose value is greater than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is implementation dependent.

- The function `fn:min()` selects an item from the input sequence `$arg` whose value is less than or equal to the value of every other item in the input sequence. If there are two or more such items, then the specific item whose value is returned is implementation dependent.

- `fn:min((xs:float(0.0E0), xs:float(-0.0E0)))` can return either positive or negative zero. XML Schema Part 2: Datatypes Second Edition [35] does not distinguish between the values positive zero and negative zero. The result is implementation dependent.

- For `fn:doc()`, its result depends entirely on the run-time environment in which the expression is evaluated. This run-time environment includes not only an unpredictable collection of resources (“the web”), but configurable machinery for locating resources and turning their contents into document nodes within the XPath data model. Both the set of resources that are reachable, and the mechanisms by which those resources are parsed and validated, are implementation dependent.

- The precise instant during the query or transformation represented by the value of `fn:current-dateTime()`, `fn:current-time()` and `fn:current-time()`.

- When casting from `xs:string` and `xs:untypedAtomic`, for `xs:anyURI`, the extent to which an implementation validates the lexical form of `xs:anyURI`.

- When casting to `xs:string` and `xs:untypedAtomic`, for data types that do not have a canonical lexical representation defined, an implementation dependent canonical representation may be used.

- When casting an `xs:float` or `xs:double` to an `xs:string` or `xs:untypedAtomic`, if more than one representation of the same value are valid, it is implementation dependent which of these representations is chosen. For example, the `xs:float` value whose exact decimal representation is 1.26743223E15 might be represented by any of the strings “1.26743223E15”, “1.26743222E15” or “1.26743224E15” (inter alia).
B.4 XSLT 2.0 and XQuery 1.0 Serialization

Implementation Defined Features

• For any implementation defined output method, it is implementation defined whether sequence normalisation process takes place.

• If the namespace URI is non-null for the method serialisation parameter, then the parameter specifies an implementation defined output method.

• The effect of additional serialisation parameters on the output of the serialiser, where the name of such a parameter must be namespace-qualified, is implementation defined or implementation dependent. The extent of this effect on the output must not override the provisions of this specification.

• The effect of providing an option that allows the encoding phase to be skipped, so that the result of serialisation is a stream of Unicode characters, is implementation defined. The serialiser is not required to support such an option.

• An serialise may provide an implementation defined mechanism to place CDATA sections in the result tree.

• If the value of the normalization-form form parameter is not NFC, NFD, NFKC, NFKD, fully-normalized, or none then the meaning of the value and its effect is implementation defined.

Implementation Dependent Features

• The actual octet order used.

• In those cases where they have no important effect on the content of the serialised result, details of the output methods defined by this specification are left unspecified and are regarded as implementation-dependent. Whether a serialise uses apostrophes or quotation marks to delimit attribute values in the XML output method is an example of such a detail.

• If the serialisation method is one of the four methods xml, html, xhtml, or text, then the additional serialisation parameters may affect the output of the serialise to the extent (but only to the extent) that this specification leaves the output implementation defined or implementation dependent.

• For characters such as > where XML defines a built-in entity but does not require its use in all circumstances, it is implementation dependent whether the character is escaped.

• If the html element is generated by an XSLT literal result element of the form

   <html xmlns="http://www.w3.org/1999/xhtml">...</html>

or by an XQuery direct element constructor of the same form, then the html element in the result document will have a node name whose prefix is “”, which will satisfy the requirements of the DTD. In other cases the prefix assigned to the element is implementation dependent.
B.5  XQuery Update Facility 1.0

Implementation Defined Features

- The revalidation modes that are supported by this implementation.
- The default revalidation mode for this implementation.
- The mechanism (if any), by which an external function can return an XDM instance and/or a pending update list to the invoking query, is implementation defined.
- The semantics of `fn:put()`, including the kinds of nodes accepted as operands by this function.

Implementation Dependent Features

- If multiple groups of nodes are inserted by multiple insert expressions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation dependent.
- If an `insert into` expression is specified without as `first` or as `last`, the positions of the inserted nodes among the children of the target node are implementation dependent.
- When processing an `upd:applyUpdates`, if as a net result of making all update primitives other than `upd:put` effective, the `children` property of some node contains adjacent text nodes, these adjacent text nodes are merged into a single text node. The string-value of the resulting text node is the concatenated string-values of the adjacent text nodes, with no intervening space added. The node identity of the resulting text node is implementation dependent.
- When processing an `upd:revalidate($top as node(), $revalidation-mode as xs:string)`, if $revalidation-mode is `lax`, define $topV as the result of the XQuery expression `validate lax {$top}`. During computation of $topV, it is necessary to maintain a mapping between each node in $topV and the corresponding node (if any) in the subtree rooted at $top (this mapping is maintained in an implementation dependent way.)
- Marking of nodes is accomplished in an implementation dependent way – for example, an implementation might maintain a list of marked nodes.
- If an implementation does not support the Update Static Typing Feature, but can nevertheless determine during the static analysis phase that an expression, if evaluated, will necessarily raise a type error at run time, the implementation may raise that error during the static analysis phase. The choice of whether to raise such an error at analysis time is implementation dependent.
C

Static Property Analysis Rules for Built-in Functions

C.1 General Rules

Here we list analysis rules that are shared by multiple functions:

• The rule BLTIN\textsuperscript{none2atom} applies to some built-in functions with zero parameters and return a sequence of atomic values:

\[
\text{Env} \vdash F() \Rightarrow (), () , \langle \eta, \mu, \sigma \rangle \quad \text{(BLTIN}\textsuperscript{none2atom})
\]

• The rule BLTIN\textsuperscript{atom2atom} applies to some built-in functions, whose parameters are all atomic typed sequences, and return a sequence of atomic values:

\[
\forall i \in 1..k; \text{Env} \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, (\bot, \bot, \bot)
\]

\[
\text{Env} \vdash F(E_1, ..., E_k) \Rightarrow () \cup \bigcup_{i=1}^{k} (\vec{r}_i \cup \vec{u}_i) , (\eta, \mu, \sigma) \quad \text{(BLTIN}\textsuperscript{atom2atom})
\]

Note that XQuery implicitly converts the values of function arguments, thus, in all rules in this section, we add a descendant::text() step to each returned path of a parameter\textsuperscript{1}.

• The rule BLTIN\textsuperscript{item2atom} applies to some built-in functions, which have one or more sequences of item() as parameters, and return a sequence of atomic values. Such functions do not access the descendants of node typed items in their parameter sequences:

\[
\forall i \in 1..k; \text{Env} \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, (\bot, \bot, \bot)
\]

\[
\text{Env} \vdash F(E_1, ..., E_k) \Rightarrow () \cup \bigcup_{i=1}^{k} (\vec{r}_i \cup \vec{u}_i) , (\eta, \mu, \sigma) \quad \text{(BLTIN}\textsuperscript{item2atom})
\]

C.2 Accessory

• \texttt{fn:node-name($arg as node()?) as xs:QName?}

\[
\text{Env} \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\bot, \bot, \bot)
\]

\[
\text{Env} \vdash \text{fn:node-name}(E_1) \Rightarrow () , \vec{r}_1 \cup \vec{u}_1 , (\eta, \mu, \sigma) \quad \text{(BLTIN}\textsuperscript{nodeName})
\]

\textsuperscript{1}In XQuery, the actual argument to a function is called an \textit{argument} and the formal argument of a function is called a \textit{parameter}. We use the same terminology here.
C.3. THE ERROR FUNCTION

- `fn:nilled($arg as node()?) as xs:boolean?`

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\bot, \bot, \bot) \\
\text{Env} & \vdash fn:nilled(E_1) \Rightarrow (\bot, \vec{r}_1, \vec{u}_1, (\eta, \mu, \sigma)) \\
\end{align*}
\]  

- `fn:string($arg as item()?) as xs:string`

To this function, the general rule `BLTIN^atom2atom` (Section C.1) applies.

- `fn:data($arg as item()*) as xs:anyAtomicType*`

To this function, the general rule `BLTIN^atom2atom` (Section C.1) applies.

- `fn:base-uri($arg as node()?) as xs:anyURI?`

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\bot, \bot, \bot) \\
\text{Env} & \vdash fn:base-uri(E_1) \Rightarrow (\bot, \vec{r}_1, \vec{u}_1, (\eta, \mu, \sigma)) \\
\end{align*}
\]  

- `fn:document-uri($arg as node()?) as xs:anyURI?`

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\bot, \bot, \bot) \\
\text{Env} & \vdash fn:document-uri(E_1) \Rightarrow (\bot, \vec{r}_1, \vec{u}_1, (\eta, \mu, \sigma)) \\
\end{align*}
\]  

C.3 The Error Function

- `fn:error($error as xs:QName?, $description as xs:string, $error-object as item()*) as none`

\[
\begin{align*}
\forall i \in 1, 3 : \text{Env} & \vdash E_i \Rightarrow \vec{r}_i, \vec{u}_i, (\bot, \bot, \bot) \\
\text{Env} & \vdash fn:error(E_1, E_2, E_3) \Rightarrow (\bot, \vec{r}_1 \cup \vec{r}_2 \cup \vec{r}_3 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{u}_3 \cup \vec{r}_1/descendant::text() \cup \vec{r}_2/descendant::text() \cup \vec{r}_3/descendant-or-self::*), (\eta, \mu, \sigma) \\
\end{align*}
\]  

C.4 The Trace Function

- `fn:trace($value as item()*, $label as xs:string) as item()*`

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\eta, \mu, \sigma) \\
\text{Env} & \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, (\bot, \bot, \bot) \\
\text{Env} & \vdash fn:trace(E_1, E_2) \Rightarrow \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1/descendant::text(), (\eta, \mu, \sigma) \\
\end{align*}
\]  

C.5 Constructor Functions

To the following function, the general rule `BLTIN^atom2atom` (Section C.1) applies:

- `fn:dateTime($arg1 as xs:date?, $arg2 as xs:time?) as xs:dateTime?`

C.6 Functions and Operators on Numerics

To all functions and operators on numerics, the general rule `BLTIN^atom2atom` (Section C.1) applies:

- `op:numeric-add($arg1 as numeric, $arg2 as numeric) as numeric`
- `op:numeric-subtract($arg1 as numeric, $arg2 as numeric) as numeric`
APPENDIX C. STATIC PROPERTY ANALYSIS RULES FOR BUILT-IN FUNCTIONS

159

- $\text{op:numeric-multiply}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as numeric}$
- $\text{op:numeric-divide}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as numeric}$
- $\text{op:numeric-integer-divide}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as xs:integer}$
- $\text{op:numeric-mod}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as numeric}$
- $\text{op:numeric-unary-plus}(\$arg \text{ as numeric}) \text{ as numeric}$
- $\text{op:numeric-unary-minus}(\$arg \text{ as numeric}) \text{ as numeric}$
- $\text{op:numeric-equal}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as xs:boolean}$
- $\text{op:numeric-less-than}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as xs:boolean}$
- $\text{op:numeric-greater-than}(\$arg1 \text{ as numeric}, \$arg2 \text{ as numeric}) \text{ as xs:boolean}$
- $\text{fn:abs}(\$arg \text{ as numeric?}) \text{ as numeric?}$
- $\text{fn:ceiling}(\$arg \text{ as numeric?}) \text{ as numeric?}$
- $\text{fn:floor}(\$arg \text{ as numeric?}) \text{ as numeric?}$
- $\text{fn:round}(\$arg \text{ as numeric?}) \text{ as numeric?}$
- $\text{fn:round-half-to-even}(\$arg \text{ as numeric?}, \$precision \text{ as xs:integer}) \text{ as numeric?}$

C.7 Functions on Strings

To all functions on string, the general rule $\text{BLTIN}^{atom\text{2atom}}$ (Section C.1) applies:

- $\text{fn:codepoints-to-string}(\$arg \text{ as xs:integer*}) \text{ as xs:string}$
- $\text{fn:string-to-codepoints}(\$arg \text{ as xs:string?}) \text{ as xs:integer*}$
- $\text{fn:compare}(\$comparand1 \text{ as xs:string?}, \$comparand2 \text{ as xs:string?}, \$collation \text{ as xs:string}) \text{ as xs:integer?}$
- $\text{fn:codepoint-equal}(\$comparand1 \text{ as xs:string?}, \$comparand2 \text{ as xs:string?}) \text{ as xs:boolean?}$
- $\text{fn:concat}(\$arg1 \text{ as xs:anyAtomicType?, \$arg2 \text{ as xs:anyAtomicType?, ...}) \text{ as xs:string}}$
- $\text{fn:string-join}(\$arg1 \text{ as xs:string*}, \$arg2 \text{ as xs:string?}, \$arg3 \text{ as xs:string}) \text{ as xs:string?}$
- $\text{fn:substring}(\$sourceString \text{ as xs:string?}, \$startingLoc \text{ as xs:double}, \$length \text{ as xs:double}) \text{ as xs:string}$
- $\text{fn:string-length}(\$arg \text{ as xs:string?}) \text{ as xs:integer}$
- $\text{fn:normalize-space}(\$arg \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:normalize-unicode}(\$arg \text{ as xs:string?}, \$normalizationForm \text{ as xs:string}) \text{ as xs:string}$
- $\text{fn:upper-case}(\$arg \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:lower-case}(\$arg \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:translate}(\$arg \text{ as xs:string?}, \$mapString \text{ as xs:string}, \$transString \text{ as xs:string}) \text{ as xs:string}$
- $\text{fn:encode-for-uri}(\$uri-part \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:iri-to-uri}(\$iri \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:escape-html-uri}(\$uri \text{ as xs:string?}) \text{ as xs:string}$
- $\text{fn:contains}(\$arg1 \text{ as xs:string?}, \$arg2 \text{ as xs:string?}, \$collation \text{ as xs:string}) \text{ as xs:boolean}$
C.8 Functions on anyURI

To the following function the general rule \( \text{BLTIN}^{\text{atom2atom}} \) (Section C.1) applies:

- \( \text{fn:resolve-uri}($relative$ as \text{xs:string}?, $base$ as \text{xs:string}) \text{ as xs:anyURI} \)

C.9 Functions and Operators on Boolean Values

To the following functions, the general rule \( \text{BLTIN}^{\text{none2atom}} \) (Section C.1) applies:

- \( \text{fn:true()} \text{ as xs:boolean} \)
- \( \text{fn:false()} \text{ as xs:boolean} \)

To the following functions, the general rule \( \text{BLTIN}^{\text{atom2atom}} \) (Section C.1) applies:

- \( \text{op:boolean-equal}($value1$ as \text{xs:boolean}, $value2$ as \text{xs:boolean}) \text{ as xs:boolean} \)
- \( \text{op:boolean-less-than}($arg1$ as \text{xs:boolean}, $arg2$ as \text{xs:boolean}) \text{ as xs:boolean} \)
- \( \text{op:boolean-greater-than}($arg1$ as \text{xs:boolean}, $arg2$ as \text{xs:boolean}) \text{ as xs:boolean} \)
- \( \text{fn:not}($arg$ as \text{item}()) \text{ as xs:boolean} \)

C.10 Functions and Operators on Durations, Dates and Times

The general rule \( \text{BLTIN}^{\text{atom2atom}} \) (Section C.1) applies to all functions and operators on durations, dates and times:

- \( \text{op:yearMonthDuration-less-than}($arg1$ as \text{xs:yearMonthDuration}, $arg2$ as \text{xs:yearMonthDuration}) \text{ as xs:boolean} \)
- \( \text{op:yearMonthDuration-greater-than}($arg1$ as \text{xs:yearMonthDuration}, $arg2$ as \text{xs:yearMonthDuration}) \text{ as xs:boolean} \)
- \( \text{op:dayTimeDuration-less-than}($arg1$ as \text{xs:dayTimeDuration}, $arg2$ as \text{xs:dayTimeDuration}) \text{ as xs:boolean} \)
- \( \text{op:dayTimeDuration-greater-than}($arg1$ as \text{xs:dayTimeDuration}, $arg2$ as \text{xs:dayTimeDuration}) \text{ as xs:boolean} \)
APPENDIX C. STATIC PROPERTY ANALYSIS RULES FOR BUILT-IN FUNCTIONS

- \texttt{op:duration-equal}(\texttt{arg1 as xs:duration}, \texttt{arg2 as xs:duration}) as \texttt{xs:boolean}
- \texttt{op:dateTime-equal}(\texttt{arg1 as xs:dateTime}, \texttt{arg2 as xs:dateTime}) as \texttt{xs:boolean}
- \texttt{op:dateTime-less-than}(\texttt{arg1 as xs:dateTime}, \texttt{arg2 as xs:dateTime}) as \texttt{xs:boolean}
- \texttt{op:dateTime-greater-than}(\texttt{arg1 as xs:dateTime}, \texttt{arg2 as xs:dateTime}) as \texttt{xs:boolean}
- \texttt{op:date-equal}(\texttt{arg1 as xs:date}, \texttt{arg2 as xs:date}) as \texttt{xs:boolean}
- \texttt{op:date-less-than}(\texttt{arg1 as xs:date}, \texttt{arg2 as xs:date}) as \texttt{xs:boolean}
- \texttt{op:date-greater-than}(\texttt{arg1 as xs:date}, \texttt{arg2 as xs:date}) as \texttt{xs:boolean}
- \texttt{op:time-equal}(\texttt{arg1 as xs:time}, \texttt{arg2 as xs:time}) as \texttt{xs:boolean}
- \texttt{op:time-less-than}(\texttt{arg1 as xs:time}, \texttt{arg2 as xs:time}) as \texttt{xs:boolean}
- \texttt{op:time-greater-than}(\texttt{arg1 as xs:time}, \texttt{arg2 as xs:time}) as \texttt{xs:boolean}
- \texttt{op:gYearMonth-equal}(\texttt{arg1 as xs:gYearMonth}, \texttt{arg2 as xs:gYearMonth}) as \texttt{xs:boolean}
- \texttt{op:gYear-equal}(\texttt{arg1 as xs:gYear}, \texttt{arg2 as xs:gYear}) as \texttt{xs:boolean}
- \texttt{op:gMonthDay-equal}(\texttt{arg1 as xs:gMonthDay}, \texttt{arg2 as xs:gMonthDay}) as \texttt{xs:boolean}
- \texttt{op:gMonth-equal}(\texttt{arg1 as xs:gMonth}, \texttt{arg2 as xs:gMonth}) as \texttt{xs:boolean}
- \texttt{op:gDay-equal}(\texttt{arg1 as xs:gDay}, \texttt{arg2 as xs:gDay}) as \texttt{xs:boolean}
- \texttt{fn:years-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:integer?}
- \texttt{fn:months-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:integer?}
- \texttt{fn:days-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:integer?}
- \texttt{fn:hours-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:integer?}
- \texttt{fn:minutes-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:integer?}
- \texttt{fn:seconds-from-duration}(\texttt{arg as xs:duration?}) as \texttt{xs:decimal?}
- \texttt{fn:year-from-dateTime}(\texttt{arg as xs:dateTime?}) as \texttt{xs:integer?}
- \texttt{fn:month-from-dateTime}(\texttt{arg as xs:dateTime?}) as \texttt{xs:integer?}
- \texttt{fn:day-from-dateTime}(\texttt{arg as xs:dateTime?}) as \texttt{xs:integer?}
- \texttt{fn:timezone-from-dateTime}(\texttt{arg as xs:dateTime?}) as \texttt{xs:dayTimeDuration?}
- \texttt{fn:year-from-date}(\texttt{arg as xs:date?}) as \texttt{xs:integer?}
- \texttt{fn:month-from-date}(\texttt{arg as xs:date?}) as \texttt{xs:integer?}
- \texttt{fn:day-from-date}(\texttt{arg as xs:date?}) as \texttt{xs:integer?}
- \texttt{fn:timezone-from-date}(\texttt{arg as xs:date?}) as \texttt{xs:dayTimeDuration?}
- \texttt{fn:hours-from-time}(\texttt{arg as xs:time?}) as \texttt{xs:integer?}
- \texttt{fn:minutes-from-time}(\texttt{arg as xs:time?}) as \texttt{xs:integer?}
- \texttt{fn:seconds-from-time}(\texttt{arg as xs:time?}) as \texttt{xs:decimal?}
- \texttt{fn:timezone-from-time}(\texttt{arg as xs:time?}) as \texttt{xs:dayTimeDuration?}
- \texttt{op:add-yearMonthDurations}(\texttt{arg1 as xs:yearMonthDuration}, \texttt{arg2 as xs:yearMonthDuration}) as \texttt{xs:yearMonthDuration}
- \texttt{op:subtract-yearMonthDurations}(\texttt{arg1 as xs:yearMonthDuration}, \texttt{arg2 as xs:yearMonthDuration}) as \texttt{xs:yearMonthDuration}
C.10. FUNCTIONS AND OPERATORS ON DURATIONS, DATES AND TIMES

- `op:multiply-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:double)` as `xs:yearMonthDuration`
- `op:divide-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:double)` as `xs:yearMonthDuration`
- `op:divide-yearMonthDuration-by-yearMonthDuration($arg1 as xs:yearMonthDuration, $arg2 as xs:yearMonthDuration)` as `xs:decimal`
- `op:add-dayTimeDurations($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration)` as `xs:dayTimeDuration`
- `op:subtract-dayTimeDurations($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration)` as `xs:dayTimeDuration`
- `op:multiply-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:double)` as `xs:dayTimeDuration`
- `op:divide-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:double)` as `xs:dayTimeDuration`
- `op:divide-dayTimeDuration-by-dayTimeDuration($arg1 as xs:dayTimeDuration, $arg2 as xs:dayTimeDuration)` as `xs:decimal`
- `fn:adjust-dateTime-to-timezone($arg as xs:dateTime?) as xs:dateTime?`
- `fn:adjust-dateTime-to-timezone($arg as xs:dateTime?, $timezone as xs:dayTimeDuration?) as xs:dateTime?`
- `fn:adjust-date-to-timezone($arg as xs:date?) as xs:date?`
- `fn:adjust-date-to-timezone($arg as xs:date?, $timezone as xs:dayTimeDuration?) as xs:date?`
- `fn:adjust-time-to-timezone($arg as xs:time?) as xs:time?`
- `fn:adjust-time-to-timezone($arg as xs:time?, $timezone as xs:dayTimeDuration?) as xs:time?`
- `op:subtract-dateTimes($arg1 as xs:dateTime, $arg2 as xs:dateTime)` as `xs:dayTimeDuration?`
- `op:subtract-dates($arg1 as xs:date, $arg2 as xs:date)` as `xs:dayTimeDuration?`
- `op:subtract-times($arg1 as xs:time, $arg2 as xs:time)` as `xs:dayTimeDuration`
- `op:add-yearMonthDuration-to-dateTime($arg1 as xs:dateTime, $arg2 as xs:yearMonthDuration)` as `xs:dateTime`
- `op:add-dayTimeDuration-to-dateTime($arg1 as xs:dateTime, $arg2 as xs:dayTimeDuration)` as `xs:dateTime`
- `op:subtract-yearMonthDuration-from-dateTime($arg1 as xs:dateTime, $arg2 as xs:yearMonthDuration)` as `xs:dateTime`
- `op:subtract-dayTimeDuration-from-dateTime($arg1 as xs:dateTime, $arg2 as xs:dayTimeDuration)` as `xs:dateTime`
- `op:subtract-yearMonthDuration-from-date($arg1 as xs:date, $arg2 as xs:yearMonthDuration)` as `xs:date`
- `op:subtract-dayTimeDuration-from-date($arg1 as xs:date, $arg2 as xs:dayTimeDuration)` as `xs:date`
• op:subtract-dayTimeDuration-from-date($arg1 as xs:date, $arg2 as xs:dayTimeDuration) as xs:date
• op:add-dayTimeDuration-to-time($arg1 as xs:time, $arg2 as xs:dayTimeDuration) as xs:time
• op:subtract-dayTimeDuration-from-time($arg1 as xs:time, $arg2 as xs:dayTimeDuration) as xs:time

C.11 Functions Related to QNames

• fn:resolve-QName($qname as xs:string?, $element as element()) as xs:QName?

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\text{Env} & \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, (\perp, \perp, \perp)
\end{align*}
\]

\[
\text{Env} \vdash \text{fn:resolve-QName}(E_1, E_2) \Rightarrow (\), \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1 / \text{descendant::text}(), (\eta, \mu, \sigma)
\]  

(BLTIn(resolveQName))

• fn:namespace-uri-for-prefix($prefix as xs:string?, $element as element()) as xs:anyURI?

\[
\begin{align*}
\text{Env} & \vdash E_1 \Rightarrow \vec{r}_1, \vec{u}_1, (\perp, \perp, \perp) \\
\text{Env} & \vdash E_2 \Rightarrow \vec{r}_2, \vec{u}_2, (\perp, \perp, \perp)
\end{align*}
\]

\[
\text{Env} \vdash \text{fn:namespace-uri-for-prefix}(E_1, E_2) \Rightarrow (\), \vec{r}_1 \cup \vec{r}_2 \cup \vec{u}_1 \cup \vec{u}_2 \cup \vec{r}_1 / \text{descendant::text}()(), (\eta, \mu, \sigma)
\]  

(BLTIn(nsuri4prefix))

• fn:in-scope-prefixes($element as element()) as xs:string*

To this function, the general rule BLTIn^item2atom (Section C.1) applies.

To the remaining functions related to QNames, the general rule BLTIn^atom2atom (Section C.1) applies:

• fn:QName($paramURI as xs:string?, $paramQName as xs:string) as xs:QName
• op:QName-equal($arg1 as xs:QName, $arg2 as xs:QName) as xs:boolean
• fn:prefix-from-QName($arg as xs:QName?) as xs:NCName?
• fn:local-name-from-QName($arg as xs:QName?) as xs:NCName?
• fn:namespace-uri-from-QName($arg as xs:QName?) as xs:anyURI?

C.12 Operators on base64Binary and hexBinary

To all operators on base64Binary and hexBinary, the general rule BLTIn^atom2atom (Section C.1) applies:

• op:hexBinary-equal($value1 as xs:hexBinary, $value2 as xs:hexBinary) as xs:boolean
• op:base64Binary-equal($value1 as xs:base64Binary, $value2 as xs:base64Binary) as xs:boolean

C.13 Operators on NOTATION

To the following function, the general rule BLTIn^atom2atom (Section C.1) applies:

• op:NOTATION-equal($arg1 as xs:NOTATION, $arg2 as xs:NOTATION) as xs:boolean
C.14 Functions and Operators on Nodes

- \texttt{fn:number($arg \text{ as } xs:anyAtomicType?) \text{ as } xs:double}

  To this function, the general rule BLTIN_{atom2atom} (Section C.1) applies.

- \texttt{fn:lang($testlang \text{ as } xs:string?, \ $node \text{ as } node()) \text{ as } xs:boolean}

  For the function \texttt{fn:lang()}, we repeat the rule \texttt{LANG} defined in Section 5.6.1 and extend it with the analysis of the \(\eta\), \(\mu\) and \(\sigma\) properties. Since the function \texttt{fn:lang()} returns a single atomic value, by Lemma 6.1.12, we have that the result of \texttt{fn:lang()} has the properties \(\eta\), \(\mu\) and \(\sigma\).

- \texttt{fn:root($arg \text{ as } node()?) \text{ as } node()?}

  We repeat the rule \texttt{ROOT} defined in Section 5.6.1 and extend it with the analysis of the \(\eta\), \(\mu\) and \(\sigma\) properties. As the function \texttt{fn:root()} returns a single node, by Lemma 6.1.12, we have that the result of \texttt{fn:root()} has the properties \(\eta\), \(\mu\) and \(\sigma\).

To the remaining functions and operators on nodes, the general rule BLTIN_{item2atom} (Section C.1) applies:

- \texttt{fn:name($arg \text{ as } node()?) \text{ as } xs:string}
- \texttt{fn:local-name($arg \text{ as } node()?) \text{ as } xs:string}
- \texttt{fn:namespace-uri($arg \text{ as } node()?) \text{ as } xs:anyURI}
- \texttt{op:is-same-node($parameter1 \text{ as } node(), \ $parameter2 \text{ as } node()) \text{ as } xs:boolean}
- \texttt{op:node-before($parameter1 \text{ as } node(), \ $parameter2 \text{ as } node()) \text{ as } xs:boolean}
- \texttt{op:node-after($parameter1 \text{ as } node(), \ $parameter2 \text{ as } node()) \text{ as } xs:boolean}

C.15 Functions and Operators on Sequences

C.15.1 General Functions and Operators on Sequences

- \texttt{fn:boolean($arg \text{ as } item()*) \text{ as } xs:boolean}

  To this function, the general rule BLTIN_{item2atom} (Section C.1) applies.

- \texttt{op:concatenate($seq1 \text{ as } item()*, \ $seq2 \text{ as } item()*) \text{ as } item()*}

  We repeat the rule \texttt{CONCAT} defined in Section 5.6.1 and extend it with the analysis of the \(\eta\) properties. As the function \texttt{op:concatenate()} returns a sequence of items, by Lemma 6.1.12, we have that the result of \texttt{op:concatenate()} has the properties \(\eta\), \(\mu\) and \(\sigma\).
APPENDIX C. STATIC PROPERTY ANALYSIS RULES FOR BUILT-IN FUNCTIONS

165

• fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType) as xs:integer*

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:index-of($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType, $collation as xs:string) as xs:integer*

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:empty($arg as item()) as xs:boolean

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:exists($arg as item()) as xs:boolean

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:distinct-values($arg as xs:anyAtomicType*) as xs:anyAtomicType*

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:distinct-values($arg as xs:anyAtomicType*, $collation as xs:string) as xs:integer*

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:unordered($sourceSeq as item()* as item()* as item()*)

To this function, the general rule BLTIN\text{atom2atom} (Section C.1) applies.

• fn:insert-before($target as item()*, $position as xs:integer, $inserts as item()*) as item()*

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow \bar{r}_1, \bar{u}_1, \langle \bot, \bot, \bot \rangle \\
\text{Env} \vdash E_2 & \Rightarrow \bar{r}_2, \bar{u}_2, \langle \bot, \bot, \bot \rangle \\
\text{Env} \vdash E_3 & \Rightarrow \bar{r}_3, \bar{u}_3, \langle \bot, \bot, \bot \rangle \\
\text{Env} \vdash \text{fn:insert-before}(E_1, E_2, E_3) & \Rightarrow \bar{r}_1 \cup \bar{r}_3, \bar{r}_2 \cup \bar{u}_1 \cup \bar{u}_2 \cup \bar{u}_3, \langle \sigma, \sigma, \sigma \rangle
\end{align*}
\]

• fn:remove($target as item()*, $position as xs:integer) as item()*

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \sigma_1 \} \\
\text{Env} \vdash E_2 & \Rightarrow \bar{r}_2, \bar{u}_2, \{ \bot, \bot, \bot \} \\
\text{Env} \vdash \text{fn:remove}(E_1, E_2) & \Rightarrow \bar{r}_1, \bar{r}_2 \cup \bar{u}_1 \cup \bar{u}_2, \{ \eta_1, \mu_1, \sigma_1 \}
\end{align*}
\]

• fn:reverse($arg as item()*) as item()*

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \sigma_1 \} \\
\text{Env} \vdash \text{fn:reverse}(E_1) & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \neg \sigma_1 \}
\end{align*}
\]

• fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) as item()*

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \sigma_1 \} \\
\text{Env} \vdash E_2 & \Rightarrow \bar{r}_2, \bar{u}_2, \{ \bot, \bot, \bot \} \\
\text{Env} \vdash E_3 & \Rightarrow \bar{r}_3, \bar{u}_3, \{ \bot, \bot, \bot \} \\
\text{Env} \vdash \text{fn:subsequence}(E_1, E_2, E_3) & \Rightarrow \bar{r}_1, \bar{r}_2 \cup \bar{u}_1 \cup \bar{u}_2 \cup \bar{u}_3, \{ \eta_1, \mu_1, \sigma_1 \}
\end{align*}
\]

• fn:unordered($sourceSeq as item()*) as item()*

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \bot \} \\
\text{Env} \vdash \text{fn:unordered}(E_1) & \Rightarrow \bar{r}_1, \bar{u}_1, \{ \eta_1, \mu_1, \emptyset \}
\end{align*}
\]

Since fn:reverse() returns the reverse of its input sequence, the \(\sigma\) property of the result of fn:reverse() is the negation of the \(\sigma\) property of its input sequence.

• fn:subsequence($sourceSeq as item()*, $startingLoc as xs:double, $length as xs:double) as item()*

Since fn:reverse() returns the reverse of its input sequence, the \(\sigma\) property of the result of fn:reverse() is the negation of the \(\sigma\) property of its input sequence.
C.15.2 Functions that Test the Cardinality of Sequences

\[ \text{Env} ⊢ E_1 \Rightarrow r_1, u_1, (\bot, \bot, \bot) \]
\[ \text{Env} ⊢ E_2 \Rightarrow r_2, u_2, (\bot, \bot, \bot) \]
\[ \text{Env} ⊢ E_3 \Rightarrow r_3, u_3, (\bot, \bot, \bot) \]

\[ \text{Env} \vdash \text{fn:deep-equal}(E_1, E_2, E_3) \Rightarrow (\bigcup_{i=1}^{3} (u_i \cup r_i) \text{descendant-or-self::} *, (\eta, \mu, \sigma)) \]

- \text{op:union}($\text{parameter1}$ as node()*)\* as node()*
  \[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash E_2 \Rightarrow r_2, u_2, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash \text{op:union}(E_1, E_2) \Rightarrow r_1 \cup r_2, u_1 \cup u_2, (\eta, \sigma, \sigma) \]

- \text{op:intersect}($\text{parameter1}$ as node()*)\* as node()*
  \[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash E_2 \Rightarrow r_2, u_2, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash \text{op:intersect}(E_1, E_2) \Rightarrow r_1 \cap r_2, u_1 \cap u_2, (\eta, \mu, \sigma) \]

- \text{op:except}($\text{parameter1}$ as node()*)\* as node()*
  \[ \text{Env} \vdash E_1 \Rightarrow r_1, u_1, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash E_2 \Rightarrow r_2, u_2, (\bot, \bot, \bot) \]
  \[ \text{Env} \vdash \text{op:except}(E_1, E_2) \Rightarrow r_1 \setminus r_2, u_1 \setminus u_2, (\eta, \mu, \sigma) \]

C.15.4 Aggregate Functions

To the following function \text{fn:count()}\*, the general rule BLTIN\text{item2atom} (Section C.1) applies.

- \text{fn:count}($\text{arg}$ as item()*) as xs:integer

To all other aggregate functions, the general rule BLTIN\text{atom2atom} (Section C.1) applies:

- \text{fn:avg}($\text{arg}$ as xs:anyAtomicType*) as xs:anyAtomicType?
- \text{fn:max}($\text{arg}$ as xs:anyAtomicType*, $\text{collation}$ as string) as xs:anyAtomicType?
- \text{fn:min}($\text{arg}$ as xs:anyAtomicType*, $\text{collation}$ as string) as xs:anyAtomicType?
- \text{fn:sum}($\text{arg}$ as xs:anyAtomicType*, $\text{zero}$ as xs:anyAtomicType?) as xs:anyAtomicType?
C.15.5 Functions and Operators that Generate Sequences

- `op:to($firstval as xs:integer, $lastval as xs:integer) as xs:integer*`

To this function, the general rule `B<sub>LT</sub>I<sub>atom2atom</sub>` (Section C.1) applies:

- `fn:id($arg as xs:string*, $node as node()) as element()*`
- `fn:idref($arg as xs:string*, $node as node()) as element()*`

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow r_1, u_1, (\bot, \bot, \bot) \\
\text{Env} \vdash E_2 & \Rightarrow r_2, u_2, (\bot, \bot, \bot) \\
\text{Env} \vdash \text{fn:id}(E_1, E_2) & \Rightarrow \bigcup_{i=1}^{r_1[\arg]/id()}.r_1 \cup r_2 \cup u_1 \cup r_2 \cup \text{descendant::text()}.\langle \eta, \emptyset, \sigma \rangle 
\end{align*}
\]

(BLT<sub>id</sub>)

For the function `fn:id()`, we repeat the rule `ID` defined in Section 5.6.1 and extend it with the analysis of the \(\eta\), \(\mu\) and \(\sigma\) properties. We omit the rule for `fn:idref()`, as it is similar to the rule `BLT<sub>id</sub>`.

- `fn:doc($uri as xs:string?) as document-node()?`

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow r_1, u_1, (\bot, \bot, \bot) \\
\text{Env} \vdash \text{fn:doc}(E_1) & \Rightarrow \text{doc}(*).r_1 \cup r_1 \cup \text{descendant::text()}.\langle \eta, \mu, \sigma \rangle 
\end{align*}
\]

(BLT<sub>doc</sub>)

For the function `fn:doc()`, we repeat the rule `DOC` defined in Section 5.6.1 and extend it with the analysis of the \(\eta\), \(\mu\) and \(\sigma\) properties.

- `fn:doc-available($uri as xs:string?) as xs:boolean`

To this function, the general rule `BLT<sub>atom2atom</sub>` (Section C.1) applies:

- `fn:collection($arg as xs:string?) as node()*`

\[
\begin{align*}
\text{Env} \vdash E_1 & \Rightarrow r_1, u_1, (\bot, \bot, \bot) \\
\text{Env} \vdash \text{fn:collection}(E_1) & \Rightarrow \text{doc}(*).r_1 \cup r_1 \cup \text{descendant::text()}.\langle \eta, \mu, \sigma \rangle 
\end{align*}
\]

(BLT<sub>collection</sub>)

C.16 Context Functions

To all context functions, the general rule `BLT<sub>none2atom</sub>` (Section C.1) applies:

- `fn:position() as xs:integer`
- `fn:last() as xs:integer`
- `fn:current-dateTime() as xs:dateTime`
- `fn:current-date() as xs:date`
- `fn:current-time() as xs:time`
- `fn:implicit-timezone() as xs:dayTimeDuration`
- `fn:default-collation() as xs:string`
- `fn:static-base-uri() as xs:anyURI`