

SANDRA MARA GUSE SCÓS VENSKE

Verificação de Tipos de XQuery: Um Protótipo em ASF+SDF

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Martin A. Musicante

CURITIBA

2004



Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

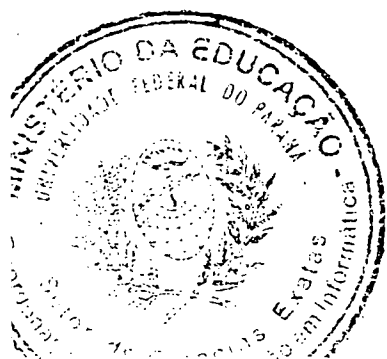
Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, da aluna *Sandra Mara Guse Scós Venske*, avaliamos o trabalho intitulado, "*VERIFICAÇÃO DE TIPOS DE XQUERY: UM PROTÓTIPO EM ASF + SDF*", cuja defesa foi realizada no dia 16 de fevereiro de 2004, às dez horas, no Auditório do Departamento de Informática da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata.

Curitiba, 16 de fevereiro de 2004.

Prof. Dr. Martin A. Musicante
DINF/UFPR - Orientador

Prof. Dr. Fernando da Fonseca de Souza
CIN/UFPE - Membro Externo

Prof. Dra. Maria Salete Marcon Gomes Vaz
UFGP - Membro Interno



*Ao meu marido, Anderson,
ao meu filho, André,
pela compreensão, apoio e paciência
em todas as horas.*

Agradecimentos

A Deus pela vida e pela força sempre presente. Sem Ele nada se pode começar nem terminar.

À toda minha família: mãe, sogros, cunhados e amigos. Pelas horas (leia-se dias) de serviços prestados como babás... Pelo apoio, incentivo e carinho.

Ao professor orientador Martin A. Musicante, que me conduziu neste trabalho. Pelas dicas certas nos lugares certos. Por me ensinar que o mais importante do trabalho é o que *eu* aprendo dele.

À Universidade Estadual do Centro-Oeste - Unicentro, que me deu suporte e dispensa para cursar o mestrado. Aos professores do Departamento de Análise de Sistemas por todo o apoio e desprendimento.

Aos professores do Departamento de Informática da Universidade Federal do Paraná pela base teórica e prática das aulas. À Universidade Federal do Paraná que, pelo Departamento de Informática, me aceitou no programa de mestrado e me deu a oportunidade da pesquisa.

Ao meu professor da graduação Ezequiel Gueiber, que me incentivou ao estudo.

A Simone Nasser Mattos e Josiane Michalak Hauagge pelas dicas na escrita da dissertação.

Ao meu marido Anderson, que sempre esteve presente com ânimo e carinho quando as forças faltavam. Porque nunca me deixou desistir ou desanimar.

Ao meu filho André pela brincadeira e pela alegria.

Conteúdo

Sumário	i
Lista de Figuras	vi
Resumo	vii
Abstract	viii
1 Introdução	1
2 Revisão Bibliográfica	4
2.1 XML	4
2.1.1 Definição de Tipo de Documento	9
2.1.2 Documentos Bem-formados	14
2.1.3 XML <i>Schema</i>	15
2.2 Linguagem de Consulta XQuery	18
2.2.1 Expressões Primárias	22
2.2.2 Expressões de Caminho	23
2.2.3 Expressões de Sequência	24
2.2.4 Expressões Aritméticas	25
2.2.5 Expressões de Comparação	26
2.2.6 Expressões Lógicas	26

2.2.7	Construtores	27
2.2.8	Expressões FLWR	28
2.2.9	Expressões de Ordem	30
2.2.10	Expressões Condicionais	31
2.2.11	Expressões Quantificadas	31
2.2.12	Expressões sobre Sequência de Tipos	32
2.2.13	Expressões de Validação	33
2.3	Outras Linguagens de Consulta para XML	33
2.4	Semântica Operacional e ASF+SDF	37
2.4.1	Semântica Operacional	37
2.4.2	Semântica de Transição	39
2.4.3	ASF+SDF	41
3	Semântica e Sistema de Tipos de XQuery	47
3.1	Modelo de Processamento	48
3.2	Modelo de Dados	49
3.3	Regras de Normalização	50
3.4	Sistema de Tipos	51
3.4.1	Valores	52
3.4.2	Tipos	54
3.4.3	Subtipos	56
3.4.4	<i>Prime Types</i>	56
3.4.5	Instruções Auxiliares	57
3.5	Semântica de Expressões	57
3.5.1	Expressões Primárias	59
3.5.2	Expressões de Caminho	61
3.5.2.1	<i>Axes</i>	61

3.5.2.2	<i>Node Tests</i>	66
3.5.3	Expressões de Sequência	71
3.5.4	Expressões Aritméticas	72
3.5.5	Construtores	74
3.5.6	Expressões FLWR	75
3.5.7	Expressões de Ordem	77
3.5.8	Expressões Condicionais	77
3.5.9	Expressões sobre Sequência de Tipos	78
3.5.10	Expressões de Validação	79
3.5.11	Observações Finais	79
4	Verificação Estática de Tipos de XQuery	81
4.1	Construção do ambiente do protótipo	81
4.2	Módulo Básico	84
4.3	Módulos Sintáticos	84
4.3.1	<i>TerminalsSyntax</i>	84
4.3.2	<i>XQueryCoreSyntax</i>	84
4.4	Módulos do Sistema de Tipos	86
4.4.1	<i>TypeSystem</i>	86
4.4.2	<i>SubTypes</i>	86
4.4.3	<i>PrimeTypes</i>	87
4.4.4	<i>TypeInteger</i>	87
4.4.5	<i>TypeBoolean</i>	87
4.5	Módulos de Ambiente	87
4.5.1	<i>StatEnvironment</i>	87
4.5.2	<i>DynEnvironment</i>	89
4.6	Módulos de Funções e Operadores	90

4.6.1	<i>FunctionsOnQNames</i>	90
4.6.2	<i>CastingTable</i>	91
4.7	Módulos de Funções Auxiliares	92
4.7.1	<i>AuxiliaryJudgments</i>	92
4.7.2	<i>AuxiliaryRules</i>	93
4.7.3	<i>SeqTypesNormalization</i>	93
4.8	Módulo de Verificação de Tipos	94
4.8.1	Expressões Primárias	96
4.8.2	Expressões de Caminho	97
4.8.3	Expressões de Sequência	107
4.8.4	Expressões Aritméticas	108
4.8.5	Construtores	111
4.8.6	Expressões FLWR	112
4.8.7	Expressões de Ordem	114
4.8.8	Expressões Condicionais	115
4.8.9	Expressões sobre Sequência de Tipos (<i>cast</i>)	116
4.8.10	Observações Finais	119
4.9	Funcionamento do Ambiente do Protótipo	119
4.10	Usando o Protótipo	120
5	Conclusões	126
5.1	Trabalhos Futuros	130
A	Arquivos de Código do Protótipo	136
A.1	Layout.sdf	136
A.2	TerminalsSyntax.sdf	136
A.3	XQueryCoreSyntax.sdf	138

A.4	TypeSystem.sdf	142
A.5	SubTypes.sdf e SubTypes.asf	147
A.6	PrimeTypes.sdf e PrimeTypes.asf	150
A.7	TypeInteger.sdf e TypeInteger.asf	154
A.8	TypeBoolean.sdf e TypeBoolean.asf	160
A.9	StatEnvironment.sdf e StatEnvironment.asf	161
A.10	DynEnvironment.sdf e DynEnvironment.asf	165
A.11	FunctionsOnQNames.sdf e FunctionsOnQNames.asf	166
A.12	CastingTable.sdf e CastingTable.asf	167
A.13	AuxiliaryJudgments.sdf e AuxiliaryJudgments.asf	179
A.14	AuxiliaryRules.sdf e AuxiliaryRules.asf	187
A.15	SeqTypesNormalization.sdf e SeqTypesNormalization.asf	190
A.16	XQueryStaticTyping.sdf e XQueryStaticTyping.asf	193

Lista de Figuras

2.1	Hierarquia de tipos de dados construídos.	15
3.1	Modelo de Processamento de XQuery.	49
4.1	Grafo do projeto.	83

Resumo

Dados semi-estruturados, e em particular XML, têm se destacado como padrão de representação de dados na *world-wide web*.

Os dados semi-estruturados apresentam vantagens para o intercâmbio de dados entre diferentes bases de dados.

Para realizar consultas a documentos (ou a bancos de dados) semi-estruturados, têm sido idealizadas linguagens de consulta. Estas linguagens seguem, em geral, o estilo da linguagem SQL.

Uma destas linguagens é XQuery, cuja semântica operacional está sendo definida pelo W3C.

Neste trabalho, propõe-se a construção de um protótipo para a verificação estática de XQuery, com base na sua semântica operacional, usando o meta-ambiente de programação ASF+SDF.

O objetivo a longo prazo do nosso trabalho é a obtenção de experiência no uso de linguagens de consulta para esta forma de representação de dados, visando o estudo de características desejáveis nesta classe de linguagens. Para isto, implementamos um protótipo, que poderá ser usado como laboratório de teste de novas características a serem adicionadas à linguagem.

Palavras-chave: Semântica Operacional, XML, XQuery.

Abstract

Semistructured data (particularly XML) are the standard data representation for information exchange in the world-wide web.

There exist a number of query languages for XML. These languages follow, in general, the SQL style. One of these languages is XQuery. The operational semantics of XQuery is being defined by W3C.

In this work, we propose the construction of a prototype for static type analysis of XQuery. The prototype implements the language's operational semantics, using the ASF+SDF meta-environment.

The long-term goal of our research is to obtain experience in the design of query languages for semistructured data, in order to study in practice the possible different characteristics of such languages. In this sence, the prototype described here is being used as a practical laboratory, in which different semantics for commands, as well as some new commands may be tested.

Keywords: Operational Semantics, XML, XQuery.

Capítulo 1

Introdução

Os dados semi-estruturados [Bun97] são usados para representar informações que não apresentam estrutura explicitamente definida ou, pelo menos, não tão rígida ou regular quanto a representação das informações nos sistemas de banco de dados tradicionais. Devido ao volume e tipo de informação disponível, a Web apresenta algumas características que correspondem às características dos dados semi-estruturados. Alguns padrões de linguagem têm surgido para representação de dados na Web, dentre os quais destaca-se o crescimento do uso da linguagem XML [ABS00, BPSMM00, BBBD00]. Existe um interesse, também crescente, em especificações de linguagens de consulta para XML. Dentre as linguagens que têm sido propostas, estão XQL [RLS98], XSL [SLR98], XML-QL [CCD⁺99], Lorel [AQM⁺97] e XQuery [BCF⁺02].

Um dos motivos para o interesse em estudar linguagens de consulta para XML se baseia na expectativa de compreender seu funcionamento e forma de estruturação e, assim, contribuir para o estudo de linguagens cada vez mais adequadas e coerentes com os dados que elas processam ou consultam. Outro fator que motivou este trabalho está no fato de que a semântica operacional de XQuery é um trabalho em desenvolvimento sendo, portanto, passível de correções e contribuições.

XQuery é uma linguagem de consulta para XML que apresenta duas importantes características: ser uma linguagem funcional e ser uma linguagem fortemente tipada. Estes aspectos contribuem para a especificação formal de XQuery e têm papel determinante na sua implementação.

O objetivo deste trabalho é o estudo da linguagem XQuery, de consulta a XML, mais especificamente de sua sintaxe e semântica, propostas no documento fornecido pelo *World Wide Web Consortium* (W3C) [DFFM02]. Este estudo foi realizado utilizando o meta-ambiente ASF+SDF, que serviu como laboratório de linguagem a fim de compreender os detalhes da construção da linguagem XQuery.

A linguagem de consulta XQuery, segundo a versão do documento de semântica formal usado [DFFM02], possui duzentas e três regras de análise de tipo estáticas, cinquenta e quatro regras de avaliação dinâmica, além de regras que são especificadas através de definições feitas no texto do documento e também através de tabelas. Todas estas regras juntas definem sua semântica formal. Para implementar o protótipo baseado em semântica operacional, cada uma destas regras semânticas foi definida no meta-ambiente ASF+SDF. Para a implementação se fez necessária uma adaptação de cada regra ao formalismo do meta-ambiente ASF+SDF. Somente após este ajuste é que as regras puderam ser utilizadas no ambiente.

Conforme o documento de especificação da linguagem [BCF⁺02], XQuery possui um conjunto de regras que determinam sua sintaxe que, em conjunto com as regras semânticas especificadas por [DFFM02], também foram fornecidas ao meta-ambiente. No entanto, foi utilizada na implementação a sintaxe representada pela gramática da linguagem *core* de XQuery [DFFM02]. XQuery *core* é a linguagem resultante da aplicação de técnicas de normalização à XQuery. Após a aplicação das regras de normalização, XQuery *core* passa a ser mais compacta que XQuery e sem redundâncias. A linguagem *core* de XQuery é um subconjunto da linguagem XQuery. A gramática fornecida na especificação da linguagem também teve que passar por transformações para se adaptar ao meta-ambiente ASF+SDF.

Esta dissertação está organizada em cinco capítulos: uma introdução, a revisão bibliográfica, o sistema de tipos de XQuery, a verificação estática de tipos de XQuery e as conclusões.

O capítulo de **Introdução** aborda algumas motivações para o desenvolvimento do trabalho.

O capítulo de **Revisão Bibliográfica** aborda conceitos relevantes sobre os temas envolvidos no trabalho. Este capítulo está dividido em quatro seções, de acordo com a seguinte ordem:

- **XML.** Aborda uma visão geral sobre dados semi-estruturados e a linguagem XML, além de conceitos pertinentes ao trabalho.

- **Linguagem de Consulta XQuery.** Esta seção aborda a estrutura e conceitos relacionados à linguagem de consulta XQuery e sua sintaxe.
- **Outras Linguagens de Consulta para XML.** Esta seção aborda algumas linguagens de consulta a XML, suas características e diferenças.
- **Semântica Operacional e ASF+SDF.** Esta seção aborda algumas definições sobre Semântica Operacional e de Transição e, em seguida, aborda o ambiente ASF+SDF, com suas características e estrutura.

O capítulo de **Semântica e Sistema de Tipos de XQuery** aborda as regras que regem a semântica operacional da verificação estática de tipos da linguagem de consulta XQuery e algumas considerações sobre elas.

O capítulo da **Verificação Estática de Tipos de XQuery** aborda a implementação do protótipo proposto e algumas considerações sobre ela.

O capítulo das **Conclusões** aborda as considerações finais, conclusões obtidas com o trabalho e sugestões de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Este capítulo aborda inicialmente uma introdução aos conceitos de XML e linguagens de consulta a XML; uma atenção especial é dada à linguagem de consulta XQuery. Em seguida são descritas noções e conceitos de Semântica Operacional e o formalismo do meta-ambiente ASD+SDF.

2.1 XML

XML (*Extensible Markup Language*) [ABS00] é um padrão desenvolvido por um grupo do *World Wide Web Consortium* (W3C) para troca de dados na Web. Este padrão complementa HTML (*Hipertext Markup Language*) [ABS00] já que foi projetado para descrever o conteúdo das páginas da Web, enquanto que HTML descreve a forma de apresentação das páginas. A forma sobre como o documento deve ser exibido deve ser fornecida em uma linguagem de especificação chamada XSL (*Extensible Stylesheet Language*) [SLR98]. Esta informação de formato deve ser incluída em uma *folha de estilo*, em um documento diferente do original, que traduzirá os dados XML para HTML. XML é uma restrição de SGML (*Standard Generalized Markup Language*) [ABS00] e os documentos XML estão em conformidade com os documentos SGML.

XML é composta por unidades de armazenamento chamadas de *elementos*. Um elemento [ABS00] é um texto delimitado por marcações (ou marcas) de início $\langle \dots \rangle$ e fim $\langle / \dots \rangle$, incluindo as marcações, e pode ter em seu interior somente texto, outros elementos ou uma combinação de ambos. As marcações são definidas pelo usuário e devem ser balanceadas, isto

é, para cada marcação deve aparecer uma marca de início e uma marca de fim. Tem-se a seguir um exemplo de documento XML:

```
<?xml version='1.0'?>
<produtos>
  <produto codigo='01a'>
    <nome> Caneta </nome>
    <quantidade> 1200 </quantidade>
    <preco> 1,3 </preco>
  </produto>
  <produto codigo='02a'>
    <nome> Lapis </nome>
    <quantidade> 2340 </quantidade>
    <preco> 0,6 </preco>
  </produto>
  <produto codigo='03b'>
    <nome> Regua </nome>
    <quantidade> 500 </quantidade>
    <preco> 0,5 </preco>
  </produto>
</produtos>
```

No exemplo, <produtos> é uma marcação de início e </produtos> é uma marcação de fim de um elemento. Todas as estruturas que figuram entre as marcas de início e fim são chamadas de *conteúdo* do elemento [ABS00]. Um elemento é dito *sub-elemento* quando possui uma relação de componente com outro elemento; no exemplo, <nome> ... </nome>, <quantidade> ... </quantidade> e <preco> ... </preco> são sub-elementos de <produto> ... </produto>.

Elementos podem ter associados a si *atributos* [BBBD00], que são pares de valores que aparecem imediatamente após o nome do elemento e devem sempre estar entre aspas. No exemplo anterior, o elemento product tem o atributo code especificado. Documentos XML podem ser representados por meio de árvore, onde os nós representam as marcações de elementos. As *entidades* [Mar00] são a forma como XML organiza fisicamente os documentos. Em algumas situações as entidades são equivalentes a arquivos; em outras, não. Para se inserir

entidades em um documento, utiliza-se *referências de entidades*. Uma referência de entidade pode ser geral ou de parâmetro.

As *referências de entidades gerais* são utilizadas em macros ou para fins de abreviação de texto. Iniciam com o caracter &, seguido por um nome e um caracter ;. Um exemplo de entidade geral é mostrado a seguir:

```
<?xml version ''1.0''?>
<!DOCTYPE catalogo [
    <!ENTITY nome ''Andre Silva'' >
]>
<catalogo>
    &nome;
</catalogo>
```

No exemplo, a entidade nome substitui o texto ''Andre Silva''.

As *referências de entidade de parâmetro* aparecem somente em Declarações de Tipo de Documento e serão vistas na Seção 2.1.1.

Algumas entidades predefinidas são:

- <: que representa o sinal de menor (<);
- &: que representa o símbolo &;
- >: que representa o sinal de maior (>);
- ': que representa o apóstrofo (');
- ": que representa as aspas (").

Por exemplo, se fosse necessário utilizar num documento XML o nome de uma empresa que possui o símbolo & em seu nome, esta seria a forma correta:

```
<empresa> Miguel Paiva &amp; Filhos </empresa>
```

já que a utilização do caracter & causaria confusão ao processador XML.

XML permite que sejam associados identificadores únicos aos elementos, que admitem a definição e utilização de *referências* fazendo com que XML possa descrever os dados sob a forma de grafos e não somente sob a forma de árvore, como mencionado anteriormente. No exemplo a seguir, o identificador `s2` é associado ao elemento `<estado>`. Depois `s2` é usado no elemento vazio `<estado-de>`, sob a forma do atributo `idref`. `Idref` faz referência ao elemento `<estado>` por intermédio do valor do atributo:

```
<?xml version='1.0'?>
<estado id='s2'>
  <ecod> PR </scode>
  <enom> Parana </snom>
</estado>
<cidade id='c2'>
  <ccod> Ct </ccode>
  <cnom> Curitiba </cnom>
  <estado-de idref='s2' />
</cidade>
```

Em XML a ordem dos elementos é importante. Por exemplo, os dois documentos abaixo não são equivalentes:

```
<?xml version='1.0'?>
<empregado>
  <nome> Jose </nome>
  <sobrenome> Pedro </sobrenome>
</empregado>
```

```
<?xml version='1.0'?>
<empregado>
  <sobrenome> Pedro </sobrenome>
  <nome> Jose </nome>
</empregado>
```

Os atributos, ao contrário, não são ordenados. Os dois documentos do exemplo abaixo são equivalentes:

```
<?xml version='1.0'?>
<empregado nome='Jose' sobrenome='Pedro' />
```

```
<?xml version='1.0'?>
<empregado sobrenome='Pedro' nome='Jose' />
```

As seções *Character Data* (CDATA) são usadas para agrupar blocos de texto com caracteres que seriam analisados por XML como marcações e que, nestas seções, são passados diretamente para a aplicação, sem interpretação. Elementos, referências ou qualquer outra marcação ficam irreconhecíveis para o analisador XML. Seções CDATA iniciam com `<[CDATA[` e terminam com `]]>`, como mostra o exemplo a seguir, baseado em [Mar00]:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<example>
<[CDATA[
  <?xml version='1.0' encoding='ISO-8859-1'?>
  <item>
    <nome> Joao Midler </nome>
    <e-mail href='mailto:jmidler@sjnet.com.br' />
  </item>
]]>
</example>
```

O exemplo utiliza uma seção CDATA para inserir um exemplo da sintaxe da linguagem XML em um documento XML.

Os comentários em XML possuem o seguinte formato:

```
<-- texto de comentario-->
```

Os comentários não fazem parte dos dados do documento, mas são parte do documento XML e podem ser passados para a aplicação. Comentários podem conter qualquer dado, com exceção de `'--'`.

As *Processing Instructions* (PIs) fornecem mecanismos para inserir, no documento XML, instruções não-XML, como *scripts*. As PIs não fazem parte dos dados caracteres do documento,

mas devem ser passadas para a aplicação. As instruções de processamento figuram entre os símbolos `<? e ?>`. Os nomes de PIs que iniciam com “XML” ou “xml” são reservados para padronização de XML. Um exemplo de utilização de instruções de processamento, baseado em [Mar00], é mostrado abaixo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="simple-ie5.xsl" type="text/xsl"?>
...
```

A primeira linha no exemplo é uma PI que já esteve presente em exemplos anteriores, que indica a declaração da versão XML utilizada. A segunda PI é utilizada para conectar folhas de estilo ao documento. O primeiro nome que aparece em uma instrução de processamento é o *destino* [Mar00], que identifica a aplicação ou dispositivo ao qual as instruções são direcionadas. As demais instruções estão em um formato específico do destino e não necessariamente precisam ser XML.

2.1.1 Definição de Tipo de Documento

Uma *Document Type Definition* (DTD) contém restrições para um tipo particular de documento XML e é colocada no início de um documento XML, antes de qualquer elemento. A DTD serve como gramática livre de contexto para o documento.

As definições de tipo de documento podem ser internas ou externas [BPSMM00]. Uma DTD é chamada interna quando apresenta as marcações de definição diretamente no documento. Uma DTD externa aponta para declarações de marcação que devem ser buscadas fora do documento. Uma DTD externa possui o formato:

```
<!DOCTYPE elemento-raiz SYSTEM endereço>
```

Sendo que *endereço* é um *Uniform Resource Locator* (URL) que aponta para a DTD. Uma DTD pode ser uma combinação de marcações internas e externas [BPSMM00].

As DTDs podem ser também uma combinação entre subconjuntos internos e externos [BBBD00].

Há quatro tipos de declarações onde as restrições sobre um documento podem ser expressas:

1. declarações de tipos de elementos
2. declarações de listas de atributos
3. declarações de entidades
4. declarações de notações

Estas declarações são detalhadas a seguir.

As **declarações de tipos de elementos** indicam os nomes dos elementos e o que eles podem conter. Uma DTD para um número variável de elementos `<peessoa/>` é mostrada abaixo, baseada em [ABS00]:

```
<!DOCTYPE db [  
    <!ELEMENT db (peessoa*)>  
    <!ELEMENT peessoa (nome, idade, email)>  
    <!ELEMENT nome (#PCDATA)>  
    <!ELEMENT idade (#PCDATA)>  
    <!ELEMENT email (#PCDATA)>  

```

O elemento raiz é `<db>` e este elemento pode ter zero ou mais sub-elementos `<peessoa>`. Cada sub-elemento `<peessoa>` possui os sub-elementos `<nome>`, `<idade>` e `<email>`, que são PCDATA. A palavra chave PCDATA (*Parsed Character Data*) se refere a um modelo de conteúdo que indica que o elemento pode conter texto. Normalmente PCDATA é usado para elementos terminais. Além de PCDATA, um modelo de conteúdo pode ser [Mar00]:

- EMPTY, que significa que o elemento é um elemento vazio;
- ANY, que significa que o elemento pode conter qualquer outro elemento declarado na DTD.

As **declarações de listas de atributos** [BPSMM00] definem o conjunto de atributos para cada elemento, que restrições de tipo se aplicam a eles e, quando pertinente, os valores *default*. Os tipos de atributos podem ser:

- *string*, que assume qualquer valor literal texto;

- um conjunto de tipos tokenizados, que possuem variações léxicas e restrições semânticas;
- tipos enumerados, que disponibilizam uma lista de valores do qual um poderá ser escolhido para atributo do elemento.

O exemplo abaixo, baseado em [ABS00], apresenta uma DTD envolvendo referências e mostra declarações de listas de atributos:

```
<!DOCTYPE geografia [
  <!ELEMENT geografia (estado cidade)*>
  <!ELEMENT estado (scod, snome, capital, cidades-em*)>
    <!ATTLIST estado id ID #REQUIRED>
  <!ELEMENT scod (#PCDATA)>
  <!ELEMENT snome (#PCDATA)>
  <!ELEMENT capital EMPTY>
    <!ATTLIST capital idref IDREF #REQUIRED>
  <!ELEMENT cidades-em EMPTY>
    <!ATTLIST cidades-em idref IDREF #REQUIRED>
  <!ELEMENT cidade (ccod, cnome, estado-de*)>
    <!ATTLIST cidade id ID>
  <!ELEMENT ccod (#PCDATA)>
  <!ELEMENT cnome (#PCDATA)>
  <!ELEMENT estado-de EMPTY>
    <!ATTLIST estado-de idref IDREF #REQUIRED>
]>
```

Os atributos permitidos são [BPSMM00, BBBD00]:

- CDATA¹, que corresponde a cadeias de caracteres;
- ID é o nome do atributo, que deve ser único para o elemento e diferente de qualquer outro no documento;
- IDREF ou IDREFS, que deve ter o valor de um único atributo ID de algum elemento do documento;

¹Não há relação entre o atributo CDATA e a seção CDATA.

- **ENTITY** ou **ENTITIES**, que deve ter o valor de uma única entidade;
- **NMTOKEN** ou **NMTOKENS**, que deve consistir de uma única palavra não necessariamente associada a outro atributo ou declaração;
- **NOTATION**, que deve se referir a uma declaração de notação em uma DTD;
- **Enumerated**, que apresenta uma lista de valores enumerados na declaração.

Os valores padrão permitidos são [BPSMM00, BBBD00]:

- **#REQUIRED**, que significa que o atributo é obrigatório, isto é, deve ter um valor explícito cada vez que o elemento aparecer;
- **#IMPLIED**, que significa que o atributo é opcional e não é fornecido um valor padrão;
- **#FIXED**, que significa que o atributo tem um valor fixo, mas não é requerido, contudo se ele ocorrer deve ter o valor especificado;
- **Default**, que significa que o valor fornecido é o padrão.

Para se fazer uma **declaração de entidades** tem-se que considerar que um documento, sua DTD e os arquivos referenciados por ele são entidades. Devido aos muitos tipos de entidades que um documento XML pode conter, as entidades podem ser classificadas em internas e externas, segundo [BPSMM00].

As entidades internas associam um nome a uma cadeia de caracteres ou texto literal e são armazenadas no documento.

As entidades externas associam um nome ao conteúdo de um arquivo, que pode conter texto ou dados binários. Se as entidades fazem referência a um arquivo que contém texto, o conteúdo do arquivo externo é inserido no documento XML e analisado como parte deste. No caso de referência a arquivo com dados binários eles podem apenas ser referenciados em um atributo e não são analisados; podem referenciar um conteúdo não-XML no documento, como figuras, por exemplo. Segue abaixo um exemplo do uso de entidades externas:

```

<?xml version ''1.0''?>
<!DOCTYPE lista [
    <!ENTITY anabeta SYSTEM ''anabeta.ent''>
    <!ENTITY anagama SYSTEM ''anagama.ent''>
]>
<lista>
    &anabeta;
    &anagama;
</lista>

```

No exemplo, alguns itens são armazenados no arquivo `anabeta.ent` e outros no arquivo `anagama.ent`, ambos referenciados por entidades. O elemento `lista` combina todos os itens em um documento, compartilhando texto. Este exemplo se refere a entidades externas gerais.

Existem as entidades externas de parâmetro, que somente podem aparecer dentro de DTDs e, portanto, devem conter marcações válidas para XML. Um exemplo é mostrado a seguir:

```

<?xml version ''1.0''?>
<!DOCTYPE endereco SYSTEM ''endereco.dtd'' [>
    <!ENTITY % pais SYSTEM ''pais.ent''>
    %pais
]>
<endereco>
    ...
</endereco>

```

No exemplo, a entidade `pais` representa uma lista de entidades para cada país, em formato de documento XML e armazenada no arquivo `pais.ent`. Esta lista é colocada dentro do documento através de uma declaração de entidade de parâmetro e faz uma referência imediata a essa entidade.

Uma notação identifica através de um nome alguma entidade que possua conteúdo não-XML, como imagens ou sons, por exemplo. Para processar tais entidades, o processador XML necessita associá-las a uma ferramenta adequada. As **declarações de notações** associam,

através de um identificador, entidades com conteúdo não-XML, a um aplicativo. Um exemplo é apresentado a seguir:

```
<!NOTATION GIF87A SYSTEM ‘‘GIF’’>
```

No exemplo, o nome de notação **GIF87A** é associado a aplicações **GIF**: ao invés disso, também poderia ser fornecido um caminho para uma aplicação disponível que processasse o tipo de arquivo desejado.

2.1.2 Documentos Bem-formados

Um documento em XML é dito bem-formado [Mar00] se possui as marcações aninhadas corretamente, atributos delimitados com aspas, entidades aceitáveis, ou seja, o documento segue a sintaxe XML. As restrições aplicadas permitem que os dados de um documento XML possam ser analisados sob a forma de uma árvore com rótulos. Um documento XML bem-formado [BPSMM00] possui as seguintes características:

- é formado por um ou mais elementos;
- possui um elemento que não é conteúdo de outro elemento - o chamado nó *raiz*;
- todo elemento *a* que possui sua marca de início contida no elemento *b*, também possui sua marca de fim contida no elemento *b*;
- obedece às restrições do documento;
- cada referência, seja direta ou indireta, feita pelo documento também é bem-formada.

Documentos bem-formados não possuem DTDs [Mar00].

Além de bem-formado, usa-se a expressão de *documento válido*, aplicada a documentos que possuem DTDs. Um documento válido é um documento que está em conformidade com sua definição de tipo de documento ou esquema, já que DTDs e esquemas definem o que cada elemento pode conter e também a estrutura geral de um documento XML.

Há duas classes de documentos (bem-formados e válidos) porque algumas aplicações não precisam de DTDs ou somente as utilizam em etapas específicas [Mar00].

2.1.3 XML Schema

Um XML Schema [BBBD00] descreve ou define a estrutura de um documento XML. XML Schema ainda está em desenvolvimento e foi criado para sanar algumas limitações que as DTDs possuem. Dentre estas limitações destaca-se que as DTDs não possuem recursos para controlar o *tipo* de informação que um elemento ou atributo pode conter. XML Schema contém um sistema de tipos de dados com o qual esta limitação pode ser corrigida. Outro ponto a ser considerado é que as DTDs não são escritas no padrão da linguagem XML, exigindo um *parser* à parte.

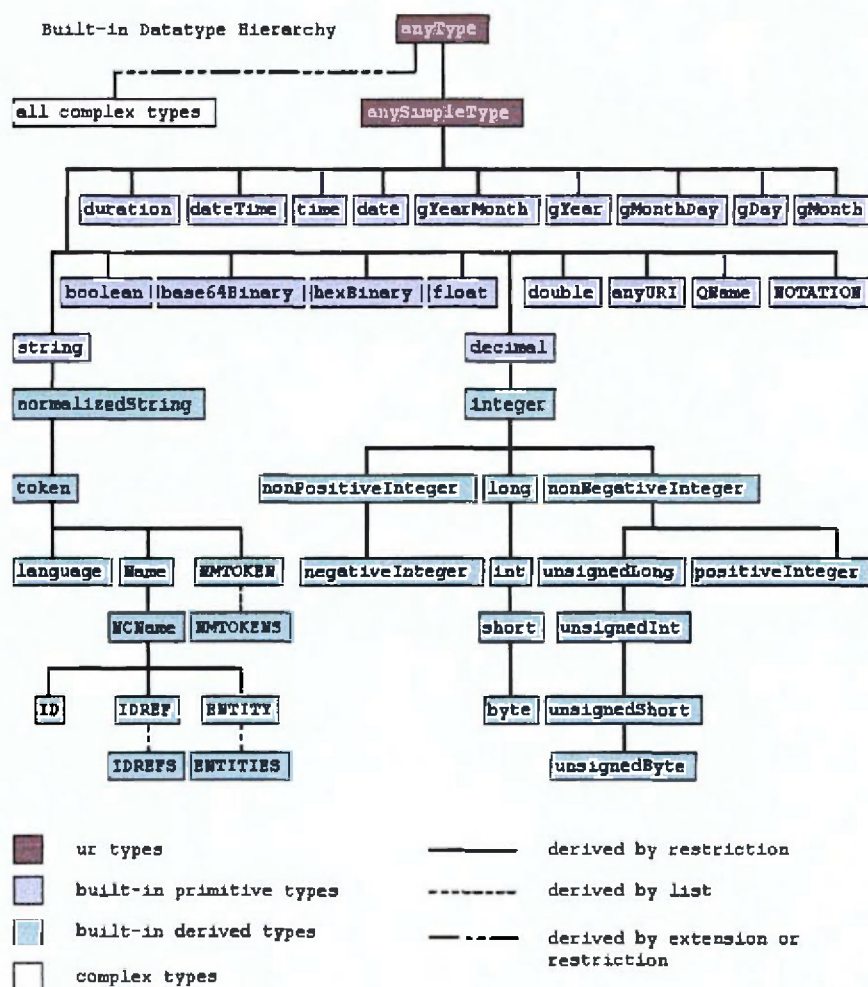


Figura 2.1: Hierarquia de tipos de dados construídos.

O propósito de um esquema [Fal01] é definir uma classe de documentos XML onde cada

documento membro desta classe está em conformidade com o esquema.

Um documento é dividido em dois tipos de conteúdo em um esquema: tipos complexos e tipos simples. Elementos que contêm somente texto ou um número são chamados de *tipos simples*. Elementos que contêm sub-elementos ou que contêm atributos são chamados de *tipos complexos*. Os atributos que contêm somente texto são considerados tipos simples.

Um conceito utilizado em XML e também em XML *Schema* é o de *namespace*. Um *namespace* [BHL99] permite que múltiplos apontadores para marcações possam existir em um mesmo documento. *Namespace* é uma coleção de nomes, identificados por uma referência a uma URL que, nos documentos XML, são usadas como tipos de elementos e como nomes de atributos. A declaração de um *namespace* usa como atributos `xmlns` ou `xmlns:`.

XML *Schema* possui recursos para definição e utilização de tipos. A Figura 2.1, extraída de [BM01] apresenta o sistema de tipos oferecido por XML Schema.

Um exemplo das definições acima, retirado de [BBBD00], é apresentado a seguir.

```
<schema targetNamespace='http://www.MyCompany.com/news'
  xmlns:news='http://www.MyCompany.com/news'
  xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
  <element name='article' type='news:articleType'/>
  <element name='dateline' type='xsd:date'/>
  <element name='body' type='xsd:string'/>
  <complexType name='articleType'>
    <element name='headline' type='news:headlineType'/>
    <element name='ByLine' type='news:ByLineType'/>
    <element ref='dateline' minOccurs='0'/>
    <element name='body' type='xsd:string'/>
    <element name='stats' type='xsd:statsType'/>
    <attribute name='section' type='xsd:string'/>
  </complexType>
  <complexType name='headlineType'>
    <element name='MainHead' type='xsd:string'/>
    <element name='SubHead' type='xsd:string'/>
  </complexType>
</schema>
```

```

</complexType>
<complexType name='ByLineType'>
  <element name='Author' type='xsd:string' />
  <element name='title' type='xsd:string' />
</complexType>
<complexType name='statsType'>
  <element name='submitted' type='xsd:date' />
  <element name='wordCount' type='xsd:decimal' />
  <element name='bureauID' type='news:BureauIDType' />
</complexType>
<simpleType name='BureauIDType' base='xsd:string'>
  <pattern value='[A-Z]-d{2}' />
</simpleType>
</schema>

```

O exemplo apresenta dois *namespaces*: *xsd*² e *news*. Inicialmente os elementos são declarados e após, são definidos. O tipo complexo *articleType* possui elementos e uma declaração de atributo.

Para propósitos de comparação apresenta-se a seguir uma DTD baseada no esquema apresentado anteriormente. Esta comparação foi retirada de [BBBD00].

```

<!DOCTYPE ARTICLE [
  <!ELEMENT article (headline, ByLine, dateline, body, stats)>
  <!ATTLIST article section CDATA 'news'>
  <!ELEMENT headline (MainHead, SubHead)>
  <!ELEMENT MainHead (#PCDATA)>
  <!ELEMENT SubHead (#PCDATA)>
  <!ELEMENT ByLine (Author, title?)>
  <!ELEMENT Author (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT stats (submitted, wordCount, bureauID)>
  <!ELEMENT submitted (#PCDATA)>

```

²Deriva de *XML Schema Document*.

```
<!ELEMENT wordCount (#PCDATA)>
<!ELEMENT bureauID  (#PCDATA)>
<!ELEMENT body      (#PCDATA)>
]>
```

Como pode ser visto nos dois últimos exemplos, o primeiro utilizando XML *Schema* e o segundo utilizando DTD, um esquema pode conter e fornecer mais informação que uma DTD. Eles podem ser mais específicos na declaração dos elementos e atributos de um documento XML, definindo a estrutura mais precisamente. Além disso, os esquemas possuem a vantagem de serem escritos em XML, enquanto as DTDs são escritas em uma linguagem completamente diferente.

2.2 Linguagem de Consulta XQuery

As informações contidas na Web possuem um formato diferente do formato dos banco de dados convencionais³. As estruturas representadas na Web não são restritas a um esquema e não possuem representação de dados e de tipos como da forma convencional; elas são irregulares e alguns campos podem aparecer ou não em determinados registros. Os dados representando estas estruturas são chamados de *dados semi-estruturados* [Bun97].

Os dados semi-estruturados, devido ao seu formato flexível, apresentam vantagens quando se quer fazer intercâmbio entre diferentes bases. Além disso, os dados semi-estruturados permitem que um usuário possa escrever uma consulta a uma base de dados sem necessariamente conhecer totalmente sua estrutura.

Para representação de dados semi-estruturados podem ser utilizadas estruturas como de árvores ou grafos. Dados semi-estruturados são *autodescritivos* [ABS00]; isto é, a *serialização*⁴ de cada item inclui não apenas os dados mas também sua descrição.

As linguagens de consulta a dados semi-estruturados, considerando a natureza irregular dos dados, também são mais flexíveis que as de sistemas de bancos de dados convencionais. Segundo

³Por exemplo, Banco de Dados Relacionais, cujos dados possuem uma estrutura fixa, pré-determinada.

⁴Conversão dos dados em um fluxo de *bytes* que pode ser facilmente transmitido e reconstruído no receptor [ABS00].

[Abi97], algumas das características que tais linguagens devem ter são:

- **Abordagem de objetos:** a flexibilidade e abstração que a noção de objetos apresenta são características que fazem com que esta abordagem complemente os dados semi-estruturados.
- **Coerção:** devido à irregularidade dos dados, algumas vezes são necessárias comparações entre tipos diferentes de dados, havendo então a necessidade de conversão implícita.
- **Expressões de caminho e padrões:** possibilitam uma expressiva melhora na navegação entre os nós da base de dados, especialmente quando fazem uso de expressões regulares e variáveis de caminho. Expressões de caminho e padrões serão vistos com mais detalhes posteriormente neste trabalho.

XML possui características como flexibilidade para a representação de diferentes tipos de informação de diferentes fontes. Devido a essas características, as linguagens de consulta para XML devem dar suporte para interpretação desse tipo de informação.

Algumas características de XML se apresentam como dificuldades para construção de linguagens de consulta, principalmente porque os bancos de dados relacionais possuem muitos princípios que XML não possui. Alguns destes princípios são [DeR98]:

- A organização dos dados possui uma hierarquia genérica - tabela, registro e campo.
- O banco de dados pode ignorar, utilizar ou modificar a ordem dos registros em uma tabela e dos campos em um registro, já que esta ordem não é parte da informação.
- Os campos em um banco de dados não têm estrutura interna, sendo unidades atômicas de tipos concretos definidos para armazenar inteiros ou datas, por exemplo.
- Em banco de dados não há uma noção de distância entre os objetos de dados.
- A ordem ou hierarquia entre dados não é, em geral, significativa.

Linguagens de consulta para os tipos de dados caracterizados acima não se aplicam para consultas sobre XML, onde os dados são hierárquicos, ordenados e com uma estrutura interna mantendo objetos de acordo com o que eles representam no mundo real.

Esta seção aborda a linguagem de consulta XQuery.

XQuery [BCF⁺02] é uma linguagem de consulta para XML que deriva da linguagem de consulta para XML chamada Quilt [CRF01]. XQuery é uma linguagem de consulta fortemente tipada, sendo que o processamento da consulta compõe-se de uma fase de verificação de tipos estática e uma fase de avaliação dinâmica. XQuery é uma linguagem funcional, o seu sistema de tipos é *sound* (consistente) e é também uma linguagem *case sensitive*, sendo que todas as suas palavras reservadas usam letras minúsculas. A especificação da linguagem XQuery é um trabalho em desenvolvimento.

XQuery tem como bloco básico as *expressões* que podem ser construídas a partir de palavras reservadas, símbolos ou operandos. Os operandos podem ser outras expressões.

O resultado de uma expressão tem como valor uma sequência ou um valor erro [BCF⁺02].

Uma *sequência* é uma coleção ordenada de zero ou mais itens. Um *item* é um valor atômico ou um nó. Um *valor atômico* é um valor no espaço de valores de um tipo atômico, como definido em XML *Schema*⁵. Um valor de nó pode ser dos seguintes tipos [FMN02]: documento, elemento, atributo, texto, *namespace*, instrução de processamento e comentário. As sequências não são aninhadas.

Um valor *erro* é um valor especial que indica que ocorreu um erro quando da avaliação da expressão.

As expressões possuem resultados, sendo que a informação que pode afetar estes resultados é chamada de *expressão de contexto*. As expressões de contexto são divididas em [BCF⁺02]:

- Contexto estático, demarcado com a informação disponível durante a análise estática da expressão.
- Contexto de avaliação, demarcado com a informação disponível durante o tempo de avaliação da expressão.

Como descrito, XQuery é uma linguagem fortemente tipada e seu sistema de tipos é baseado em XML *Schema* [Fal01, BM01]. A verificação de tipos oferece uma *verificação de tipos estática*.

⁵Um tipo simples que não é uma lista de tipos nem uma união de tipos.

feita em tempo de compilação, e a *verificação de tipos dinâmica*, feita em tempo de execução da consulta.

Para se fazer referência a um tipo em uma expressão de XQuery, a sintaxe apresentada a seguir é utilizada. Esta sintaxe de produções é chamada de *Seqüência de Tipos*, porque ela descreve o tipo de um valor de XQuery, que é uma seqüência.

$$\begin{aligned}
SequenceType &::= (ItemType OccurrenceIndicator) \mid \text{"empty"} \\
ItemType &::= ((\text{"element"} \mid \text{"attribute"}) ElemOrAttrType?) \\
&\mid \text{"node"} \mid \text{"processing - instruction"} \\
&\mid \text{"comment"} \mid \text{"text"} \mid \text{"document"} \\
&\mid \text{"item"} \mid AtomicType \mid \text{"untyped"} \mid \text{"atomic"} \text{"value"} \\
ElemOrAttrType &::= (QName (SchemaType \mid SchemaContext?)) SchemaType \\
SchemaType &::= \text{"of"} \text{"type"} QName \\
SchemaContext &::= \text{"in"} SchemaGlobalContext (\text{"/" } SchemaContextStep) * \\
SchemaGlobalContext &::= QName \mid (\text{"type"} QName) \\
SchemaContextStep &::= QName \\
AtomicType &::= QName \\
OccurrenceIndicator &::= (\text{"*"} \mid \text{"+"} \mid \text{"?"})?
\end{aligned}$$

A seguir são apresentados alguns exemplos de seqüência de tipos, retirados de [BCF⁺02], que podem ser usados em expressões de XQuery ou como parâmetros de funções:

- **xs:date** se refere ao tipo construído *date* de XML Schema;
- **attribute?** se refere a um atributo opcional;
- **element office:letter** se refere a um elemento com nome específico;
- **item*** se refere a uma seqüência de zero ou mais nós (*node*) ou valores atômicos.

As expressões podem ser primárias, de caminho, de seqüência, aritméticas, de comparação, lógicas, construtores, FLWR⁶ (iniciais de *For*, *Let*, *Where* e *Return*), de ordem, condicionais,

⁶Pronunciadas como *flower*.

quantificadas, sobre tipos de dados e de validação. Nas seções seguintes é apresentada uma descrição breve de cada uma delas.

2.2.1 Expressões Primárias

As **expressões primárias** [BCF⁺02] são representadas pelas primitivas básicas de uma linguagem e são os literais, as variáveis, as chamadas de função e a utilização dos parênteses para determinar a precedência de operadores. As produções de gramáticas que definem as expressões primárias são:

$$\begin{aligned} \textit{PrimaryExp} ::= & \textit{Literal} \mid \textit{FunctionCall} \mid (\textit{"\$"} \textit{VarName}) \\ & \mid \textit{ParenthesizedExpr} \end{aligned}$$

A regra para definição de variáveis (**VarName**) é:

$$\textit{VarName} ::= \textit{QName}$$

A regra para definição de literais (**Literal**) é:

$$\begin{aligned} \textit{Literal} &::= \textit{NumericLiteral} \mid \textit{StringLiteral} \\ \textit{NumericLiteral} &::= \textit{IntegerLiteral} \mid \textit{DecimalLiteral} \mid \textit{DoubleLiteral} \\ \textit{IntegerLiteral} &::= \textit{Digits} \\ \textit{DecimalLiteral} &::= (\textit{"." Digits}) \mid (\textit{Digits "." [0-9]*}) \\ \textit{DoubleLiteral} &::= ((\textit{"." Digits}) \mid (\textit{Digits "." [0-9]*}?)) ([\textit{e} \mid \textit{E}]) ([\textit{+} \mid \textit{-}])? \textit{Digits} \\ \textit{StringLiteral} &::= ([\textit{""}][^\textit{""}]) * [\textit{""}] \mid ([\textit{'"}][^\textit{'"}]) * [\textit{''}] \\ \textit{Digits} &::= [0-9]^+ \end{aligned}$$

A regra para definição de expressões entre parênteses (**ParenthesizedExpr**) é:

$$\textit{ParenthesizedExpr} ::= \textit{"(" ExprSequence? ")"}$$

A regra para definição de chamadas de funções (*FunctionCall*) é:

$$FunctionCall ::= QName "(" (Expr ("," Expr)*)? ")"$$

A regra para definição de comentários (*ExprComment*) é:

$$ExprComment ::= "{ - - " [!]* " - - }"$$

2.2.2 Expressões de Caminho

As **expressões de caminho** [BCF⁺02] são um recurso que XQuery dispõe para percorrer e endereçar partes específicas de um documento XML através do uso de expressões que localizam os nós na árvore do documento. Estas expressões são avaliadas no contexto de avaliação. A gramática para expressões de caminho é apresentada a seguir:

$$PathExpr ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr) \\ RelativePathExpr$$

$$RelativePathExpr ::= StepExpr ("/" "//") StepExpr *$$

$$StepExpr ::= (ForwardStep | ReverseStep | PrimaryExpr) | Predicates$$

$$ForwardStep ::= (ForwardAxis NodeTest) | AbbreviatedForwardStep$$

$$ReverseStep ::= (ReverseAxis NodeTest) | AbbreviatedReverseStep$$

$$Predicates ::= ("[" Expr "]") *$$

$$ForwardAxis ::= \langle "child" " :: " \rangle \\ | \langle "descendant" " :: " \rangle \\ | \langle "attribute" " :: " \rangle \\ | \langle "self" " :: " \rangle \\ | \langle "descendant - or - self" " :: " \rangle$$

$$ReverseAxis ::= \langle "parent" " :: " \rangle$$

$$NodeTest ::= KindTest | NameTest$$

$$NameTest ::= QName | Wildcard$$

$$\begin{aligned}
\textit{Wildcard} &::= "*" \mid \langle \textit{NCName} ":" "*" \rangle \mid \langle "*" ":" \textit{NCName} \rangle \\
\textit{KindTest} &::= \textit{ProcessingInstructionTest} \mid \textit{CommentTest} \\
&\quad \mid \textit{TextTest} \mid \textit{AnyKindTest} \\
\textit{ProcessingInstructionTest} &::= \textit{"processing-instruction"} "(" \textit{StringLiteral?} ")" \\
\textit{CommentTest} &::= \textit{"comment"} "(" ")" \\
\textit{TextTest} &::= \textit{"text"} "(" ")" \\
\textit{AnyKindTest} &::= \textit{"node"} "(" ")" \\
\textit{AbbreviatedForwardStep} &::= "." \mid ("@" \textit{NameTest}) \mid \textit{NodeTest} \\
\textit{AbbreviatedReverseStep} &::= ".."
\end{aligned}$$

Um exemplo de expressão de caminho é mostrado a seguir:

```
document('livros.xml')/acervo/livro
```

O resultado desta consulta contém dois elementos <acervo> e <livro>. O método `document` retorna o nó raiz de "livros.xml" e o percurso pela árvore segue pelo elemento <acervo> e, a partir deste, sobre todos os elementos <livro> (que são filhos do elemento <acervo>).

2.2.3 Expressões de Seqüência

Expressões de seqüência [BCF⁺02] são as construções e combinações de seqüências através de operadores que XQuery oferece suporte.

A *construção* de uma seqüência [BCF⁺02] pode ser através de uma expressão parentizada ou de uma expressão de série de valores. A expressão parentizada é construída com zero ou mais expressões separadas por vírgulas e delimitadas por parênteses. A expressão de série de valores pode ser usada para construir uma seqüência de valores inteiros consecutivos, utilizada com o operador `to`. A gramática para construção de uma seqüência é apresentada a seguir:

$$\begin{aligned}
\textit{ExprSequence} &::= \textit{Expr} ("," \textit{Expr}) * \\
\textit{RangeExpr} &::= \textit{AdditiveExpr} ("to" \textit{AdditiveExpr}) *
\end{aligned}$$

A seguir, num exemplo de expressão parentizada, as seqüências 9, (1, 20) e (8, 1) são adicionadas a uma expressão:

(9, (1, 20), (8, 1))

Segue também um exemplo que usa uma expressão de valores numa expressão de seqüência:

(4, 5 to 9)

A *combinação* de seqüências [BCF⁺02] utiliza vários operadores para combinação de seqüências de nós. Alguns deles são os operadores de *união*, *intersecção* e *exceção*, sendo que eles sempre tomam como operandos dois nós. O operador de *união* retorna uma seqüência contendo todos os nós que ocorrem em um dos operandos. O operador de *intersecção* retorna uma seqüência contendo todos os nós que ocorrem em ambos os operandos. O operador de *exceção* retorna uma seqüência contendo todos os nós que aparecem no primeiro operando, mas não aparecem no segundo. A gramática para combinação de uma seqüência é apresentada a seguir:

$$\begin{aligned} UnionExpr &::= IntersectExceptExpr (("union" \mid "|") IntersectExceptExpr) * \\ IntersectExceptExpr &::= UnaryExpr (("intersect" \mid "except") UnaryExpr) * \end{aligned}$$

2.2.4 Expressões Aritméticas

Os operadores para criação de **expressões aritméticas** [BCF⁺02] fornecidos por XQuery são: adição (+), subtração (-), multiplicação (*), divisão (div) e módulo (mod), além das formas unárias para sinal positivo (+) e negativo (-). Convém observar que o operador de subtração (-) deve ser precedido de um espaço em branco se não estiver sendo utilizado somente com valores numéricos. Isto evita que a expressão seja avaliada como um simples *token* como, por exemplo, em a-b. A gramática para expressões aritméticas é apresentada a seguir:

$$\begin{aligned} AdditiveExpr &::= MultiplicativeExpr (("+" \mid "-") MultiplicativeExpr) * \\ MultiplicativeExpr &::= UnionExpr (("*" \mid "div" \mid "idiv" \mid "mod") UnionExpr) * \\ UnaryExpr &::= ("-" \mid "+") * ValueExpr \\ ValueExpr &::= ValidateExpr \mid CastExpr \mid Constructor \mid PathExpr \end{aligned}$$

2.2.5 Expressões de Comparação

As **expressões de comparação** [BCF⁺02] permitem quatro tipos de comparação:

1. Comparações de valores, utilizadas para valores simples, utilizando os operadores *eq* (igualdade), *ne* (desigualdade), *lt* (menor que), *le* (menor ou igual que), *gt* (maior que), *ge* (maior ou igual que) entre duas expressões.
2. Comparações gerais, definidas adicionando semântica existencial às comparações dos valores. Utiliza os operadores *=*, *!=* (desigualdade), *<*, *<=*, *>*, *>=* entre duas expressões. Por exemplo, a comparação geral entre as seqüências A e B sob a forma *A < B* é *verdadeira* se a comparação de valores *a lt b* é *verdadeira* para algum item a em A e para algum item b em B. Caso contrário, a comparação geral *A < B* é *falsa*.
3. Comparações de nós utiliza os operadores *is* e *isnot* entre duas expressões, para avaliar a relação de igualdade entre dois nós.
4. Comparações de ordem fazem comparações entre nós e utilizam os operadores *<<* (ocorre antes), *>>* (ocorre depois), *precedes* (precede sem ser ancestral) e *follows* (segue sem ser descendente) entre duas expressões.

A gramática para expressões de comparação é apresentada a seguir:

```
ComparisonExpr ::= RangeExpr ((ValueComp | GeneralComp
                               | NodeComp | OrderComp) RangeExpr)?
ValueComp      ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
GeneralComp    ::= "=" | "!=" | "<" | "<=" | ">" | ">="
NodeComp       ::= "is" | "isnot"
OrderComp      ::= "<<" | ">>"
```

2.2.6 Expressões Lógicas

Uma **expressão lógica** [BCF⁺02] é uma expressão de conjunção (*and*) ou uma expressão de disjunção (*or*). Pode ainda ser utilizada a função de negação (*not*). O valor de uma expressão

lógica é sempre verdadeiro (*true*) ou falso (*false*), salvo nos casos em que um valor de erro apareça. Um exemplo de expressão com retorno de valor de erro é:

`1 = 1 and 3 div 0 = 47`

A gramática para expressões lógicas é apresentada a seguir:

$$\begin{aligned} OrExpr &::= AndExpr \text{ (``or'' } AndExpr) * \\ AndExpr &::= UnorderedExpr \text{ (``and'' } UnorderedExpr) * \end{aligned}$$

2.2.7 Construtores

Construtores [BCF⁺02] podem criar estruturas XML através de consultas. Há construtores para elementos, atributos, seções CData, instruções de processamento e comentários.

Construtores são definidos como produções de gramática da seguinte forma:

$$\begin{aligned} Constructor &::= ElementConstructor \mid XmlComment \\ &\mid XmlProcessingInstruction \mid CdataSection \\ &\mid ComputedDocumentConstructor \\ &\mid ComputedElementConstructor \\ &\mid ComputedAttributeConstructor \\ ElementConstructor &::= (\text{ (``<'' } QName AttributeList \text{ (``/ >''} \\ &\mid \text{ (``>'' } ElementContent * \text{ ``</'' } QName S? \text{ ``>''} ) ) \\ ElementContent &::= Char \mid \text{ ``{'' } ``}'' \mid ElementConstructor \\ &\mid EnclosedExpr \mid CdataSection \mid CharRef \\ &\mid PredefinedEntityRef \mid XmlComment \\ &\mid XmlProcessingInstruction \\ AttributeList &::= ( S ( QName S? \text{ ``= '' } S? AttributeValue)?) * \\ AttributeValue &::= ([`` ( EscapeQuot \\ &\mid AttributeValueContent ) * [ ``]) \\ &\mid ([' (EscapeApos \end{aligned}$$

$$\begin{aligned}
& | \textit{AttributeValueContent}) * [']) \\
\textit{AttributeValueContent} & ::= \textit{Char} \mid \textit{CharRef} \mid \text{"\{"} \mid \text{"\}" } \\
& | \textit{EnclosedExpr} \mid \textit{PredefinedEntityRef} \\
\textit{EnclosedExpr} & ::= (\mid \text{"\("}) \textit{ExprSequence} \text{"\)" } \\
\textit{ComputedElementConstructor} & ::= ((\text{"element"} \textit{QName} \text{"\("}) \\
& | (\text{"element"} \text{"\("}) \textit{Expr} \text{"\)" } \text{"\("}) \textit{ExprSequence}? \text{"\)" } \\
\textit{ComputedAttributeConstructor} & ::= ((\text{"attribute"} \textit{QName} \text{"\("}) \\
& | (\text{"attribute"} \text{"\("}) \textit{Expr} \text{"\)" } \text{"\("}) \textit{ExprSequence}? \text{"\)" } \\
\textit{ComputedDocumentConstructor} & ::= \text{"document"} \text{"\("} \textit{ExprSequence} \text{"\)" } \\
\textit{CdataSection} & ::= \text{"<![CDATA["} \textit{Char} * \text{"\]}>" } \\
\textit{XmlProcessingInstruction} & ::= \text{"<?"} \textit{PITarget} \textit{Char} * \text{"?>" } \\
\textit{XmlComment} & ::= \text{"<!--"} \textit{Char} * \text{"-->" }
\end{aligned}$$

2.2.8 Expressões FLWR

As expressões FLWR [BCF⁺02], apresentam quatro cláusulas:

1. **for**: associa uma ou mais variáveis a expressões, criando tuplas de variáveis *bindings* vindas do produto cartesiano da sequência de valores para os quais as expressões avaliam.
2. **let**: relaciona uma variável diretamente a um expressão inteira. Se uma cláusula **for** estiver presente, a variável *bindings* gerada pela cláusula **let** é adicionada às tuplas geradas pela cláusula **for**; caso contrário, a cláusula **let** gera uma tupla com todas as variáveis *bindings*.
3. **where**: pode ser usada como um filtro para as tuplas de variável *bindings* produzidas pelas cláusulas **for** e **let**.
4. **return**: contém uma expressão que é utilizada na construção do resultado da expressão FLWR.

Expressões FLWR são definidas como produções de gramática da seguinte forma:

$$\begin{aligned}
 FLWRExpr &::= ((ForClause \mid LetClause) + WhereClause? \text{ "return" }) * QuantifiedExpr \\
 ForClause &::= \text{ "for" } TypeDeclaration? \$ VarName \text{ "in" } \\
 &\quad Expr (\text{ "," } TypeDeclaration? \$ VarName \text{ "in" } Expr) * \\
 LetClause &::= \text{ "let" } TypeDeclaration? \$ VarName \text{ " := " } \\
 &\quad Expr (\text{ "," } TypeDeclaration? \$ VarName \text{ " := " } Expr) * \\
 WhereClause &::= \text{ "where" } Expr \\
 TypeDeclaration &::= SequenceType
 \end{aligned}$$

O exemplo seguinte, retirado de [BCF⁺02], mostra o uso de expressões FLWR para transformar uma bibliografia de entrada em uma lista de autores:

```

<authlist>
{let $input := document("bib.xml")}
  for $a in distinct-values($input//author)
  return
    <author>
      {<name>
        { $a/text() }
      }
    </name>
    <books>
      {for $b in input//book
        where $b/author = $a
        return $b/title}
      }
    </books>}
  </author>}
</authlist>

```

O resultado da expressão do exemplo acima pode ser como o apresentado a seguir.

```

<authlist>

```

```

<author>
  <name>
    W. Stevens
  </name>
  <books>
    <title> TCP/IP Illustrated </title>
    <title> Advanced Programming in the Unix Environment </title>
  </books>
</author>
</authlist>

```

2.2.9 Expressões de Ordem

As **expressões de ordem** [BCF⁺02] fornecem maneiras de controlar a ordem dos itens numa sequência. Expressões de ordem são definidas como produções de gramática da seguinte forma:

$$\begin{aligned}
 \textit{SortExpr} &::= \textit{OrExpr} \left(\textit{"stable"}? \textit{"sort"} \textit{"by"} \textit{"("} \textit{SortSpecList} \textit{")"} \right) \\
 \textit{SortSpecList} &::= \textit{Expr} \textit{SortModifier} \left(\textit{","} \textit{SortSpecList} \right)? \\
 \textit{SortModifier} &::= \left(\textit{"ascending"} \mid \textit{"descending"} \right)? \left(\textit{"empty"} \textit{"greatest"} \mid \textit{"empty"} \textit{"least"} \right)? \\
 \textit{UnorderedExpr} &::= \textit{"unordered"}? \textit{FLWRExpr}
 \end{aligned}$$

O valor da expressão *Expr*, que aparece antes da palavra chave **sortby** é chamado de *sequência de entrada*. O resultado da aplicação de expressões de ordem é chamado de *sequência de saída*. As expressões que aparecem do lado direito da palavra chave **sortby** são as *expressões ordenadas* e podem ser seguidas das palavras chaves **ascending** ou **descending**, sendo que **ascending** é o valor assumido se nenhum outro valor for especificado. Os opcionais **empty greatest** e **empty least** definem a posição de sequências vazias retornadas por expressões ordenadas.

2.2.10 Expressões Condicionais

As expressões condicionais [BCF⁺02] de XQuery são baseadas nas palavras reservadas *if*, *then* e *else* e possuem a seguinte forma como produções de gramática:

$$IfExp ::= ("if" "(" Expr ")" "then" Expr "else") * Instanceof Expr$$

A expressão depois da palavra reservada *if* é chamada de *expressão de teste*, enquanto que as expressões depois das palavras reservadas *then* e *else* são chamadas *expressões de resultado*. Um exemplo é apresentado a seguir:

```
if ($widget1/unit-cost < $widget2/unit-cost)
then $widget2
else $widget1
```

No exemplo acima a expressão de teste é uma expressão de comparação. O primeiro passo do processamento de uma expressão condicional é a avaliação da expressão de teste.

2.2.11 Expressões Quantificadas

Nas expressões quantificadas [BCF⁺02] que XQuery oferece suporte podem aparecer os quantificadores existencial e universal, sendo que o valor de uma expressão desse tipo é sempre verdadeiro (*true*) ou falso (*false*). Uma expressão quantificada possui a forma:

$$\begin{aligned} QuantifiedExp ::= & ((\text{"some"} \mid \text{"every"}) TypeDeclaration? \$ VarName \text{"in"} \\ & (\text{","} TypeDeclaration? \$ VarName \text{"in"} Expr) * \text{"satisfies"}) * \\ & TypeswitchExpr \end{aligned}$$

Se o quantificador *some* for utilizado, a expressão quantificada é *true* se pelo menos uma avaliação da expressão de teste tem o valor booleano *true*; em caso contrário, a expressão quantificada é *false*. Um exemplo é apresentado a seguir.

```
some $var in (1, 2, 4) satisfies $var * 2 = 4
```

No exemplo, a expressão quantificada retorna *true* porque o teste para a expressão retorna *true* para um caso.

Utilizando o quantificador *every*, a expressão quantificada é *true* se toda avaliação da expressão de teste tem o valor booleano *true*. Um exemplo é apresentado a seguir.

```
every $var in (1, 2, 4) satisfies $var * 2 = 4
```

No exemplo, a expressão quantificada retorna *false* porque o teste para a expressão retorna *false* para pelo menos um caso.

2.2.12 Expressões sobre Seqüência de Tipos

As expressões sobre tipos de dados [BCF⁺02] possuem ocorrências de tipos de seqüências que, por sua vez, ocorrem em alguns tipos de expressões de XQuery, como pode ser visto na gramática a seguir.

InstanceofExpr ::= *ComparisonExpr* (“instance” “of” *SequenceType*)

TypeswitchExpr ::= (“typeswitch” (“*Expr*”) *CaseClause* +
“default” (“\$” *VarName*)? “return”) * *IfExpr*

CaseClause ::= “case” *SequenceType* (“\$” *VarName*)? “return” *Expr*

CastExpr ::= (“cast” “as” | “treat” “as”) *SequenceType* *ParenthesizedExpr*

O operador booleano *instance of* retorna *true* se o valor do operando do lado esquerdo combina com o tipo do operando do lado direito; caso contrário, retorna *false*. Por exemplo:

```
$x instance of element of type animal
```

Se o valor associado a variável *\$x* é um elemento cujo conteúdo é um valor do tipo *animal*, o exemplo retorna *true*.

O exemplo seguinte, retirado de [BCF⁺02], mostra o uso de uma expressão *typeswitch*:

```
typeswitch ($animal)
  case element duck return quack($animal)
```

```

case element dog return woof($animal)
case element wolf return woo($animal)
default return ‘‘No sound’’

```

2.2.13 Expressões de Validação

As expressões de validação [BCF⁺02] são expressas como produções de gramática da seguinte forma:

$$ValidateExpr ::= \text{“validate” } SchemaContext? \{“Expr”\}$$

O seguinte exemplo cria e valida um elemento `person` declarado globalmente:

```

validate {
  <person>
    <name>
      <first> Elvira </first>
      <last> Fischbein </last>
    </name>
  </person>
}

```

Uma expressão de validação pode conter um `SchemaContext` que é utilizado na validação de elementos e atributos declarados localmente. As produções para `SchemaContext` foram definidas no início da Seção 2.2.

2.3 Outras Linguagens de Consulta para XML

Além de XQuery, outras linguagens de consulta foram projetadas para XML. Dentre elas são comentadas brevemente nesta seção LOREL, XML-QL, XML-GQL, XSL e XQL.

A linguagem de consulta **LOREL** (*Lightweight Object Repository Language*) [AQM⁺97] é uma linguagem que foi estendida para dados XML, já que foi projetada originalmente para

consulta a dados semi-estruturados. LOREL é considerada uma linguagem amigável, especialmente para usuários do estilo SQL, e possui um forte mecanismo de coerção e consultas mais concisas através de atalhos sintáticos.

A linguagem de consulta **XML-QL** (*XML Query Language*) [DFF⁺99, ABS00] é uma linguagem que utiliza padrões para extrair informações dos dados dos documentos XML. Dentre suas características é uma linguagem declarativa, completa em termos relacionais, extrai os dados de documentos XML e constrói novos documentos e pode trabalhar com visões ordenadas ou não sobre um documento XML. XML-QL tem suporte a consulta, integração, construção e transformação de dados XML.

A linguagem de consulta **XML-GL** (*XML Graphical Language*) [CCD⁺99] é uma linguagem gráfica de consulta a documentos XML, tanto para o conteúdo de documentos quanto para suas DTDs. A linguagem de consulta pode ser aplicada a um documento ou a um conjunto de documentos e produz como resultado um novo documento XML.

A linguagem de consulta **XSL** (*XML Extensible Stylesheet Language*) [SLR98] é essencialmente uma linguagem de transformação de documentos XML para HTML, sendo responsável pela apresentação de um documento XML. XSL, contudo, devido a suas características de linguagem de transformação, pode funcionar como linguagem de consulta, mas não é completa em termos relacionais e possui limitações. Uma vantagem é que o modelo de dados de XSL corresponde ao de XML. Um programa XSL é um conjunto de regras de modelo, cada regra consistindo de um padrão, que é comparado com os nós da árvore. Um modelo é instanciado para formar parte da árvore de resultado.

A linguagem de consulta **XQL** (*XML Query Language*) [RLS98] é uma linguagem para endereçamento e filtragem de texto e elementos de documentos XML. XQL é uma extensão do padrão de sintaxe XSL, acrescentando, por exemplo, lógica booleana, filtros e indexação de conjunto de nós, e foi projetada para ser mais simples e concisa que XSL.

Uma análise comparativa entre as cinco linguagens de consulta caracterizadas anteriormente pode ser encontrada em [BC00]. A Tabela 2.3, retirada de [BC00], sumariza algumas destas comparações; a coluna sobre XQuery não consta da tabela original sendo acrescentada baseada nas especificações de XQuery descritas neste trabalho. Algumas observações sobre as carac-

	LOREL	XML-QL	XML-Gl	XSL	XQL	XQuery
Modelo de dados específico	Sim	Sim	Sim	Sim	Não	Sim
<i>Joins</i>	Sim	Sim	Sim	Não	Não	Sim
Semântica de resultado	OIDs	Doc	Doc	Doc	Doc	Doc
Expressão de caminho	Sim	Sim	Parcial	Sim	Sim	Sim
Quantificação existencial	Sim	Sim	Sim	Sim	Sim	Sim
Quantificação universal	Sim	Não	Não	Não	Não	Sim
Negação	Sim	Não	Sim	Sim	Sim	Não
Construção de novos elem.	Sim	Sim	Sim	Sim	Não	Sim
Construção de grupos	Sim	Não	Sim	Não	Não	Sim
Agregação	Sim	Não	Sim	Parcial	Parcial	Não
Aninhamento de consultas	Sim	Sim	Não	Sim	Sim	Sim
Ordenação do resultado	Sim	Sim	Sim	Sim	Não	Sim
Coerção de tipos	Sim	Não	Não	Não	Parcial	Sim
Linguagem de atualização	Sim	Não	Sim	Não	Não	Não

Tabela 2.1: Tabela de comparação entre algumas linguagens de consulta a XML (Fonte: [BC00]).

terísticas avaliadas, também baseadas em [BC00], são feitas a seguir:

- **Modelo de dados específico:** avalia se a linguagem de consulta introduz seu próprio modelo de dados ou se segue as características do modelo de dados de XML.
- **Joins:** avalia se a linguagem de consulta possui o recurso de comparação entre dois ou mais atributos ou dados XML, no mesmo documento ou em documentos diferentes.
- **Semântica de resultado:** avalia o que a linguagem de consulta apresenta como resultado da consulta (um novo documento XML ou ponteiros para os identificadores de objetos dos nós (OIDs)).
- **Expressões de caminho:** avalia se linguagem de consulta usa expressões de caminho, acrescentadas de coringas e expressões regulares.
- **Quantificação existencial:** avalia se linguagem de consulta possui algum recurso que atue como predicado existencial. Um predicado existencial sobre um conjunto de instâncias é satisfeito se pelo menos uma instância satisfaz o predicado.
- **Quantificação universal:** avalia se linguagem de consulta possui algum recurso que atue como predicado universal. Um predicado universal sobre um conjunto de instâncias é satisfeito se todas as instâncias satisfazem o predicado.
- **Negação:** avalia se linguagem de consulta possui algum recurso que atue como negação de um predicado. A negação de um predicado sobre um conjunto de instâncias é satisfeita se nenhuma das instâncias satisfazem o predicado.
- **Construção de novos elementos:** avalia se linguagem de consulta pode criar um novo elemento XML através de mecanismos de consulta.
- **Construção de grupos:** avalia se linguagem de consulta pode agregar os elementos do resultado da consulta para funções especiais.
- **Agregação:** avalia se linguagem de consulta utiliza funções de agregação, que calculam um valor escalar a partir de vários conjuntos de valores.

- **Aninhamento de consultas:** avalia se linguagem de consulta pode compor consultas com subconsultas aninhadas.
- **Ordenação do resultado:** avalia se linguagem de consulta pode ordenar o resultado da consulta (ascendente ou descendente) utilizando algum dado do resultado.
- **Coerção de tipos:** avalia se linguagem de consulta está hábil a comparar valores de diferentes tipos e se converte implicitamente alguns tipos de dados diferentes.
- **Linguagem de atualização:** avalia se linguagem de consulta oferece suporte inserção, exclusão e atualização de elementos.

2.4 Semântica Operacional e ASF+SDF

A *Semântica Operacional* é uma maneira de expressar a forma como uma linguagem de programação age quando da execução de um programa. ASF+SDF (*Algebraic Specification Formalism + Syntax Definition Formalism*) é um ambiente de desenvolvimento interativo para manipulação de textos escritos em uma linguagem formal; ele recebe como entrada a sintaxe e semântica da linguagem e produz um ambiente para ela.

Nesta seção são abordados conceitos e formalismos que caracterizam a semântica operacional e o ambiente ASF+SDF.

2.4.1 Semântica Operacional

A introdução descrita nesta seção está baseada em [Mus96] e [Ast91].

Uma linguagem de programação pode expressar seu significado especificando o comportamento dos programas gerados por ela. O funcionamento de um programa pode ser definido pela descrição de suas partes: comandos, declarações, expressões, entre outras. Algumas das abordagens para expressar a semântica de linguagens de programação são: *Semântica Denotacional* [Wat91, Sch86], *Semântica de Ações* [Wat91] e *Semântica Operacional* [Ast91]. O conjunto de regras que especificam a estrutura de uma linguagem de programação correspondem a *sintaxe*

da linguagem e permitem determinar se os programas escritos naquela linguagem estão no formato correto. Há duas formas de apresentar a sintaxe de uma linguagem: sintaxe abstrata e sintaxe concreta [ASU88]. A sintaxe abstrata define a estrutura de uma linguagem com uma descrição de alto nível. Na descrição da sintaxe concreta de uma linguagem são descritos tanto os elementos da linguagem quanto a forma em que estes são obtidos.

A *Semântica Denotacional* é um formalismo para a descrição semântica [Wat91, Sch86] que inicialmente era conhecido como *Semântica Matemática* por se basear em fundamentos matemáticos. Neste paradigma, a semântica de qualquer parte de um programa - suas declarações, comandos, expressões - é dita sua *denotação*. A semântica denotacional define para cada entidade da linguagem um objeto matemático e uma função que relaciona instâncias dos dois. O método é chamado denotacional porque os objetos denotam o significado de suas entidades sintáticas correspondentes.

A *Semântica de Ações* é um formalismo desenvolvido para prover descrições compreensíveis de aplicações e linguagens da vida real [Wat91]. A semântica de ações foi desenvolvida a partir da *Semântica Denotacional* [Sch86], combinando formalidade com aspectos pragmáticos desejáveis, tais como: facilidade de leitura, pois faz uso de termos utilizados na linguagem informal; modularidade; capacidade de tratar abstrações e facilidade para provar propriedades algébricas para produzir semânticas equivalentes. A semântica de ações utiliza entidades *ad-hoc* que são chamadas de ações e que são operacionais, processando informações gradualmente.

A *Semântica Operacional* expressa a forma *como* uma linguagem de programação age quando da execução de um programa. A construção da definição operacional de uma linguagem é feita organizando-a em sintaxe abstrata e um conjunto de regras de avaliação das frases da linguagem (com funções recursivas, por exemplo).

Uma característica padrão de todos os formalismos operacionais é a definição indutiva de uma *relação de transição*, onde é definido o significado das partes sintáticas. A semântica operacional utiliza definições indutivas [Acz77] e sistemas de transição [Ast91]

Como exemplo, apresenta-se a sintaxe abstrata, com representação infixa com parênteses, para expressões *Exp*. Assume-se que *Num* denota um conjunto de representações de números naturais; *true* e *false* são símbolos sintáticos para verdade e falsidade; *Id* e *Bid* denotam dois

conjuntos de símbolos, o conjunto de identificadores inteiros e booleanos, respectivamente. Considera-se ainda n , id e bid como variáveis sobre Num , Id e Bid , respectivamente. Os conjuntos $Iexp$ e $Bexp$ representam as expressões inteiras e booleanas da linguagem Exp . Usando a notação BNF (*Backus-Naur Form*), tem-se:

$$\begin{aligned}
\langle Exp \rangle &::= \langle Iexpr \rangle \mid \langle Bexpr \rangle \\
\langle Iexpr \rangle &::= \langle Num \rangle \mid \langle Id \rangle \mid \langle (Iexp \ iop \ Iexp) \rangle \\
\langle Bexpr \rangle &::= \langle true \rangle \mid \langle false \rangle \mid \langle Bid \rangle \mid \\
&\quad \langle (Bexp \ bop \ Bexp) \rangle \mid \langle (Iexp \ rop \ Iexp) \rangle \mid \langle \neg(Bexp) \rangle \\
\langle iop \rangle &::= * \mid + \mid - \\
\langle bop \rangle &::= \vee \mid \wedge \\
\langle rop \rangle &::= > \mid < \mid \leq \mid \geq
\end{aligned}$$

A notação acima indicada corresponde à definição indutiva da linguagem Exp . O nome à esquerda do símbolo $::=$ indica o conjunto a ser definido.

Para definir a semântica da linguagem Exp , pode-se utilizar a Semântica Indutiva Estrutural (SIS), que pode ser feita de duas formas [Ast91]:

- semântica de passo grande (*big step*);
- semântica de passo pequeno (*small step*), também chamada Semântica de Transição.

Neste trabalho aborda-se somente a semântica de transição devido ao fato de que o protótipo implementado foi desenvolvido utilizando o meta-ambiente ASF+SDF [BK03], que utiliza conceitos de semântica de transição para a criação de um ambiente operacional relativo a uma determinada linguagem.

2.4.2 Semântica de Transição

A introdução abordada esta seção está baseada em [Mus96] e [Ast91].

A *semântica de transição* é um formalismo que permite a definição de uma relação de um passo (*step*) entre as frases de uma linguagem. A semântica de um programa é caracterizada

pela relação *step* e seu fecho reflexivo e transitivo. Para definir um sistema correspondente para avaliação de expressões da esquerda para a direita, separa-se a avaliação da expressão em uma seqüência de avaliações parciais - *small step* - sendo que cada avaliação parcial corresponde a *um passo* de avaliação elementar (por exemplo, avaliação de uma variável, adição de dois inteiros). Cada passo é transformado em uma expressão dentro de outra, através da reescrita de sub-expressões elementares dentro de um valor básico, ou seja, um elemento que denote alguma representação (por exemplo, dos valores inteiros). Os estágios intermediários quando da avaliação de uma expressão são descritos pela introdução de novas expressões que não são elementos da linguagem.

Considerando a linguagem *Exp* abordada na Seção 2.4.1, assume-se que a seguinte expressão deve ser avaliada:

$$((3 - 7) * 5)$$

Para que a avaliação ocorra, a expressão deve passar por uma atribuição de valores para conjuntos finitos de identificadores, sempre preservando o tipo - o que é chamado de *state* (estado). O conjunto de estados denotados por *State* é o conjunto de mapeamentos finitos de subconjuntos de:

- $Id \cup Bid$, se o estado $s \in State$;
- $s(id) \in Int$, se $id \in Id$;
- $s(bid) \in Bool$, se $bid \in Bid$;

Sendo que *Int* e *Bool* denotam representação de valores inteiros e booleanos, respectivamente. Então, a semântica de uma expressão inteira e , $I[e]$ é uma função que toma o estado s devolvendo um valor inteiro $I[e/s]$ (um elemento de *Int*). A semântica de expressões booleanas é feita de forma análoga.

Enfim, a avaliação da expressão anterior, em um estado s , é obtida:

$$\begin{aligned} ((\underline{3} - \underline{7}) * \underline{5}) &\xrightarrow{s} ((3 - 7) * \underline{5}) \xrightarrow{s} \\ ((3 - 7) * \underline{5}) &\xrightarrow{s} (-4 * \underline{5}) \xrightarrow{s} (-4 * 5) \xrightarrow{s} -20 \end{aligned}$$

Para considerar um exemplo de expressões booleanas, a semântica de transição usa *regras* para realizar a avaliação. Para cada possível passo de uma dada expressão, uma regra é definida e a definição dos passos é baseada na estrutura sintática da linguagem. Uma parte pequena da semântica para expressões booleanas que completa o exemplo é dada a seguir:

$$id \xrightarrow{s} s(id) \quad (2.1)$$

$$bid \xrightarrow{s} s(bid) \quad (2.2)$$

$$\frac{e_1 \xrightarrow{s} e'_1}{(e_1 \text{ iop } e_2) \xrightarrow{s} (e'_1 \text{ iop } e_2)} \quad (2.3)$$

As regras acima são divididas em duas categorias:

- regras como a 2.1 e 2.2 que definem um passo de avaliação elementar e são chamadas de *axiomas*;
- regras como a 2.3 que definem a transformação de uma expressão correspondente aplicando *um passo de avaliação* elementar e são chamadas de *regras de inferência*. O que está acima da linha é chamado de *premissa* e o que está abaixo da linha é a *conclusão*.

Cada regra corresponde a um passo de avaliação elementar e transforma uma expressão em outra pela reescrita de suas subexpressões.

2.4.3 ASF+SDF

O ASF+SDF [BK03] é um ambiente de desenvolvimento interativo para manipulação de textos escritos em uma linguagem formal. Ele combina dois formalismos: o *Algebraic Specification Formalism* (ASF) e o *Syntax Definition Formalism* (SDF) [BK03, HHKR92]. O ASF+SDF recebe como entrada a sintaxe e semântica da linguagem e produz um ambiente para ela.

O meta-ambiente ASF+SDF oferece [BK03]:

- um formalismo de especificação algébrica que se baseia em lógica equacional;

- estruturação modular de especificações;
- definição integrada de sintaxe abstrata, léxica e livre de contexto;
- facilidade para o usuário definir sua própria notação em suas especificações;
- integração entre formalismos de sintaxe e semântica;
- um editor de especificação dirigido pela sintaxe;
- um mecanismo de teste das especificações;
- ferramentas como um *parser*, *pretty printer* e um interpretador ou compilador.

ASF [BK03, HHKR92] baseia-se na definição da sintaxe abstrata das funções e num conjunto de equações condicionais que representam a semântica de tais funções, enquanto que SDF [BK03, HHKR92] permite a definição de sintaxe concreta⁷ e abstrata e fornece a tradução de textos para árvores sintáticas abstratas. A combinação destes formalismos permite ao ASF+SDF integrar sintaxe e semântica numa sequência de declarações de módulos.

A estrutura de um módulo⁸, [BK03], é apresentada a seguir:

```
module <ModuleName>
  <ImportSection>*
  <ExportOrHiddenSection>*
equations
  <ConditionalEquation>*
```

Pode-se ver que um módulo é formado por um cabeçalho, seguido de uma seção *imports* repetida zero ou mais vezes, seguida de uma seção *exports* ou *hiddens* repetida zero ou mais vezes, seguida pela seção de *equations* que apresenta as equações condicionais.

A seção *imports* introduz um ou mais módulos externos que farão parte da especificação. A seção *exports* ou *hiddens* descreve definição da gramática, sendo que *exports* faz com que as

⁷Sintaxe léxica e sintaxe livre de contexto [ASU88].

⁸O módulo é aqui representado como estrutura unitária apenas por conveniência técnica. Normalmente as seções de sintaxe e *equations* são feitas em estruturas físicas (arquivos) distintas.

entidades apresentadas nela sejam visíveis a outros módulos, enquanto que a seção *hiddens* faz com que as entidades sejam locais ao módulo [BK03].

A definição da gramática de cada seção do módulo é composta pelos seguintes itens:

- *Imports*: iniciado pela palavra reservada `imports` seguida pelo nome de um ou mais módulos.
- *Aliases*: iniciado pela palavra reservada `aliases`. Os *aliases* são nomes dados a valores constantes ou literais para fins de abreviação destes valores.
- *Sorts*: iniciado pela palavra reservada `sorts`. Nesta seção devem ser declarados os nomes de símbolos não-terminais, terminais e classes de caracteres.
- *Lexical syntax*: iniciado pelas palavras reservadas `lexical syntax`. Esta seção descreve os *tokens* léxicos, que são as menores estruturas de um texto.
- *Context-free syntax*: iniciado pelas palavras reservadas `context-free syntax`. Esta seção contém a descrição da estrutura sintática concreta e abstrata das sentenças da linguagem.
- *Priorities*: iniciado pela palavra reservada `priorities`. Esta seção define as prioridades e associatividades para as sentenças, especialmente para evitar ambigüidades.
- *Variables*: iniciado pela palavra reservada `variables`. Esta seção consiste de uma lista de nomes de variáveis seguidas por um símbolo.

Algumas destas seções são exemplificadas no módulo a seguir, baseado num exemplo de [BK03]:

```
module Test
imports Layout
exports
  sorts ID DECL TYPE
  lexical syntax
    [a-z]+ -> ID
```

```

context-free syntax
    'decl' {ID ' ',''}+ '::' TYPE -> DECL
    'integer' -> TYPE
    'real' -> TYPE

hiddens
variables
    'Id' -> ID
    'Type' [0-9]* -> TYPE
    'Id-list' [\']* -> {ID ' ',''}*
    'Id-ne-list' -> {ID ' ',''}+

```

A semântica da linguagem objeto da especificação é definida mediante equações. Na seção `equations` [BK03] é adicionada a semântica para as funções declaradas nas seções `lexical` `syntax` e `context-free syntax`. As equações consistem de dois termos que podem conter variáveis livres.

Existem dois tipos de equações: incondicionais e condicionais.

As **equações incondicionais** [BK03] são *equations* com dois termos L e R, indicando lados esquerdo e direito⁹ da equação, respectivamente. L e R são da mesma classe, as variáveis que aparecem em R aparecem em L e L não é uma variável simples. Segue um exemplo de equação incondicional:

[TagId] L = R

Sendo que TagId é um rótulo para a equação e inicia com uma letra ou dígito, seguido de letras, dígitos e/ou o sinal de menos. As equações incondicionais são um caso especial de equações condicionais.

Uma **equação condicional** [BK03] é uma igualdade¹⁰ associada a uma ou mais condições. As opções de sintaxe do ASF+SDF para uma equação condicional são:

(1) [TagId] L = R when C1, C2, ...

(2) [TagId] C1, C2, ... ==> L = R

⁹Iniciais de *Lefthand* e *Righthand*, em inglês.

¹⁰A igualdade pode ser chamada de *conclusão* da equação condicional.

(3)[TagId] C1, C2, ...

=====

L = R

Sendo que C1, C2, ... são as condições positivas ou negativas, avaliadas da esquerda para a direita. O exemplo seguinte contém a seção *equations* com equações incondicionais. Ele foi retirado de [vD].

```
module Booleans
imports Layout
exports
  sorts BOOL
  context-free syntax
    true          -> BOOL
    false         -> BOOL
    BOOL 'and'    -> BOOL {left}
    BOOL 'or'     -> BOOL {left}
    'not' BOOL    -> BOOL
    '(', 'BOOL(') -> BOOL {bracket}
  priorities
    'or' < 'and' < 'not'
  variables
    P          -> BOOL
  equations
    [1] true or P = true
    [2] false or P = P
    [3] true and P = P
    [4] false and P = false
    [5] not true = false
    [6] not false = true
```

No exemplo anterior, a seção *context-free syntax* determina que as conjunções e disjunções devem ser associadas da esquerda para a direita. Isto é feito através da palavra reservada *left* à direita da sentença. A definição de funções léxicas ou livres de contexto podem ser seguidas de

atributos, que definem propriedades adicionais da função. Os atributos relacionados à sintaxe são:

- *bracket*, que permite a definição de parêntesis que auxiliam na determinação das prioridades dos operadores nas expressões;
- *left*, *right*, *non-assoc*, *assoc*, que definem associatividade das funções;
- *prefer*, que indica que a função que possui este atributo deve ser preferida sobre outras funções (sem este atributo) em caso de ambigüidade sintática;
- *avoid*, que indica que a função deve ser usada como última opção em caso de ambigüidade sintática;
- *reject*, que pode ser usado para proibir explicitamente certas construções sintáticas.

A seção *priorities* do exemplo estabelece que a negação (*not*) tem prioridade sobre a conjunção (*and*) que tem prioridade sobre a disjunção (*or*).

O conjunto de constantes e funções introduzido pela seção *context-free syntax* do exemplo constitui a *signature* [vD] e é a partir deste conjunto que se pode construir os termos. As igualdades sobre os termos são obtidas através das *equations*, que podem ser usadas para *reescrever* de termos complexos em termos mais simples. O processo de aplicação sucessiva das regras de *reescrever* até que um termo não possa mais ser simplificado reduz um termo a sua *forma normal*.

Capítulo 3

Semântica e Sistema de Tipos de XQuery

Este capítulo aborda a semântica formal da verificação de tipos da linguagem de consulta XQuery e está baseado no documento fornecido pelo W3C [DFFM02]. A semântica formal de XQuery é um trabalho em desenvolvimento.

Dois aspectos importantes de XQuery regem a semântica formal da linguagem. Eles são [DFFM02]:

1. **XQuery é uma linguagem funcional.** Como a maioria das construções da linguagem são expressões (e expressões podem ser compostas arbitrariamente), o resultado de uma expressão pode ser usado como entrada para outra expressão. Outro aspecto é que as variáveis são passadas sempre por valor.
2. **XQuery é uma linguagem tipada.** Os tipos podem vir de um ou vários XML *Schemas* e XQuery pode realizar operações baseada nestes tipos. XQuery possui suporte à análise estática de tipos, o que permite a detecção de erros antes da execução da consulta. O sistema de tipos de XQuery é baseado no sistema de Tipos fornecido por XML *Schema*, abordado na Seção 2.1.3, embora não ofereça suporte a todas as características de XML *Schema*.

Apresenta-se a seguir a semântica formal de XQuery [DFFM02] e a notação utilizada para

expressar a semântica.

3.1 Modelo de Processamento

O *modelo de processamento* para avaliação de consultas é baseado em quatro fases, sendo que cada fase utiliza o resultado da fase anterior e gera sua saída para ser utilizada pela fase seguinte. As fases são:

1. *Parsing*. Esta fase verifica a sintaxe da consulta em XQuery¹ e pode apontar erros. Se não ocorrerem erros, algum tipo de representação interna da árvore gramatical é criada.
2. Normalização. Durante esta fase, cada consulta é mapeada para sua versão equivalente em *XQuery Core Language*. A linguagem *core* é um subconjunto da linguagem XQuery e é composta por versões simplificadas e sem resundâncias de consultas feitas em XQuery. Após a normalização, as duas últimas fases podem fornecer a semântica completa da consulta somente fornecendo uma semântica da consulta normalizada em *core*. As regras de normalização serão vistas com mais detalhes na Seção 3.3.
3. Análise estática de tipos. Esta fase verifica se cada consulta em *core* possui tipos adequados e determina seus tipos estáticos. A análise estática de tipos é executada no sentido *bottom-up* na aplicação das regras de inferência de tipos sobre as expressões. Se ocorrer alguma incompatibilidade entre os tipos, um *erro estático* pode resultar; caso nenhum erro ocorra, uma árvore sintática é gerada para cada subexpressão com seu tipo estático, além de ser fornecido um tipo de saída ao resultado da consulta.
4. Avaliação dinâmica. Nesta fase, o resultado da consulta é calculado. A semântica da avaliação é definida somente para termos de consulta em *core*. A avaliação é executada no sentido *bottom-up* na aplicação das regras de inferência de tipos sobre as expressões, iniciando com a avaliação de literais.

As fases 1, 2 e 3 são chamadas de passos *em tempo de compilação* ou *em tempo de análise*. A fase 4 é chamada de passo *em tempo de execução* ou *run-time*.

¹A sintaxe de XQuery foi apresentada de um modo geral na Seção 2.2.

A fases 2, 3 e 4 serão descritas para fins da semântica de XQuery através de *regras de inferência*, que fornecem uma notação para descrever a forma como a semântica de uma expressão deriva da semântica de suas subexpressões (especificação *bottom-up*).

A figura 3.1 mostra o modelo de processamento de XQuery. Este trabalho implementa, como descrito no Capítulo 4, a fase de **Inferência de Tipos e Verificação de Tipos**, algumas pequenas partes da fase de **Normalização XQuery** e a base para a fase de **Processamento XQuery**.

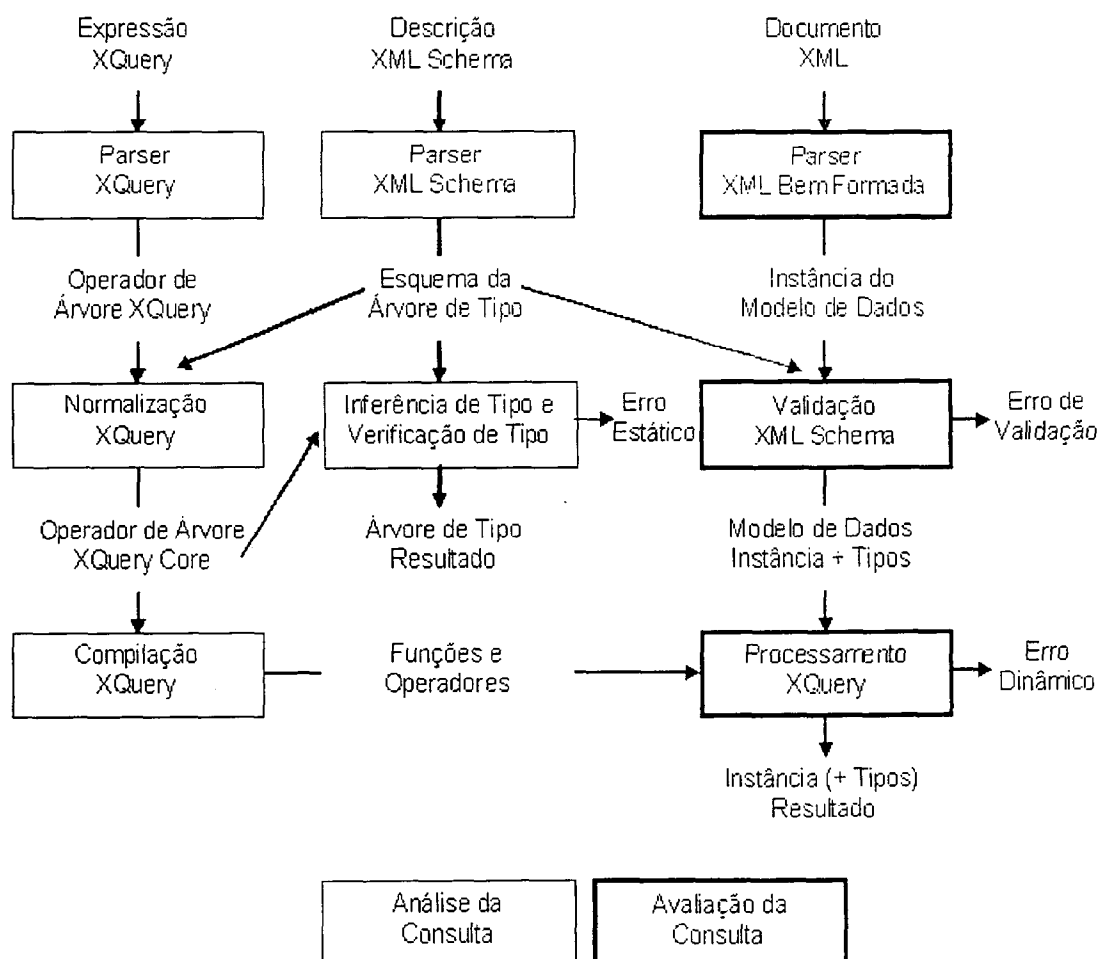


Figura 3.1: Modelo de Processamento de XQuery.

3.2 Modelo de Dados

Os componentes básicos do modelo de dados de XQuery são:

- **Error**, que é um valor de erro simples;
- **SimpleValue**, que é um valor qualquer do espaço de valores dos tipos simples de XML *Schema*, como definido na Seção 2.1.3;
- **Node**, que é um nó, podendo ser dos seguintes tipos [FMN02]: **DocumentNode** (documento), **ElementNode** (elemento), **AttributeNode** (atributo), **TextNode** (texto), **NamespaceNode** (*namespace*), **PINode** (instrução de processamento) e **CommentNode** (comentário).
- **Sequence**, que é uma coleção ordenada de nós, uma coleção ordenada de valores tipados ou uma coleção ordenada de itens.
- **SchemaComponent**, que é o tipo de um nó.

Em XQuery são considerados dois tipos de ordenação: a ordem de seqüência e a ordem de documento. As seqüências de itens no modelo de dados de XQuery são ordenadas; *ordem de seqüência* refere-se à ordem destes itens em uma determinada seqüência. A *ordem de documento* se refere à ordem geral entre todos os nós em um dado documento.

A notação `dm:error()` se refere a um valor de erro no modelo de dados.

3.3 Regras de Normalização

A *Normalização* é definida como um conjunto de regras de reescrita, que determinam a transformação de expressões de XQuery para XQuery *core*. A semântica estática é apresentada como *regra de inferência de tipos*, onde as expressões de XQuery são relacionadas a tipos e é determinado sob que condições uma expressão é bem tipada. A semântica dinâmica é apresentada de forma operacional como *regra de inferência de valores*, que especifica a ordem na qual um expressão de XQuery é avaliada e relaciona expressões de XQuery a valores.

A normalização aplica uma *função de tradução* a cada expressão na sintaxe de XQuery e retorna uma expressão na sintaxe de XQuery *core*. A notação

[*Expr*]

denota o resultado da transformação de uma expressão *Expr* em XQuery em uma expressão em XQuery *core*. As regras de normalização seguem, portanto, esta forma [DFFM02]:

$$\begin{array}{c} [Expr] \\ == \\ [coreExpr] \end{array}$$

A expressão acima do sinal de $==$ denota uma expressão na linguagem XQuery antes da tradução e *coreExpr* denota a expressão equivalente em XQuery *core*.

Este trabalho implementa as regras de normalização para *SequenceType* somente. Estas regras foram implementadas por serem utilizadas diretamente na análise estática de algumas expressões. As outras regras de reescrita de XQuery em XQuery *core* podem ser implementadas da mesma forma (tradução *fonte-a-fonte*).

3.4 Sistema de Tipos

O *sistema de tipos* de XQuery é usado na especificação da sua semântica estática e dinâmica, além de ser usado para descrever a semântica de expressões sobre seqüência de tipos e regras de conversão de tipos. Ele também é a base para a semântica estática de XQuery.

Algumas correções e modificações foram feitas no sistema de tipos de XQuery para a implementação do protótipo. Dentre elas, podem ser citadas e resumidas:

- A inclusão do valor *error* como regra de produção para os não-terminais *Value* e *Type*. Esta inclusão se justifica pelo fato de que ao longo da especificação das regras que compõem a análise de tipos de XQuery *core* (de acordo com [DFFM02]) o valor de erro é utilizado sem, contudo, fazer parte da gramática da linguagem. O tratamento de erros para XQuery ainda não está definido pelo W3C.
- A inclusão do não-terminal *Wildcard* como regra de produção para os não-terminais *ElementName* e *AttributeName*. Este não-terminal foi adicionado devido a inconsistências

nas regras de análise de tipo estática para *node tests* (Seção 4.8.2). Tais regras aplicavam *wildcards* para *ElementName* e *AttributeName*, sem isto ser permitido, contudo, na gramática destes não-terminais.

- A unificação de indicadores de ocorrências (*OccurrenceIndicator*). O sistema de tipos de XQuery e a gramática normalizada de XQuery tinham não-terminais definidos distintamente para indicadores de ocorrências. Estas definições eram semelhantes e, por isso, foram alteradas e unificadas em um só não-terminal, que passou a ser utilizado em toda a aplicação.
- A incorporação dos não-terminais *AttributeAll* e *ElementContent* ao sistema de tipos de XQuery. Estes dois não-terminais, embora possuindo produções, não faziam parte do sistema de tipos e nem da gramática de XQuery devido ao fato de que não eram chamados como produções de nenhum outro não-terminal. Eles são usados nas regras que definem o funcionamento das instruções auxiliares (Seção 3.4.5). Nestas regras são usados como se fossem parte da definição de tipos (*ItemType*). Este uso esclareceu sua origem e justificou sua inclusão como produção do não-terminal *ItemType*.
- Inclusão do não-terminal *Type* como produção de *PrimeType*. As regras semânticas que determinam a especificação de *prime types* são definidas utilizando as produções do não-terminal *Type* sem, contudo, este estar associado a *prime types*. Devido a este fato, o não-terminal *Type* foi incluído como produção de *PrimeType*.

As subseções a seguir abordam as notações formais que descrevem os valores e tipos de XQuery, subtipos e *prime types*. Comentários adicionais são feitos sobre as correções e modificações.

3.4.1 Valores

As seguintes produções de gramática apresentam a notação formal para valores:

$$\begin{aligned}
 \textit{Value} &::= \textit{Item} \mid (\textit{Value} \text{ ", " } \textit{Value}) \mid \text{ "(" " } \mid \text{ "error"} \\
 \textit{SimpleValue} &::= \textit{AtomicValue} \mid (\textit{SimpleValue} \text{ ", " } \textit{SimpleValue}) \mid \text{ "(" " }
 \end{aligned}
 \tag{3.1}$$

ElementValue ::= “element” *ElementName* *TypeAnnotation*? “”*Value*“” (3.2)

AttributeValue ::= “attribute” *AttributeName* *TypeAnnotation*? “”*Value*“” (3.3)

DocumentValue ::= “document” “”*Value*“”

TextValue ::= “text” “”*String*“”

NodeValue ::= *ElementValue* | *AttributeValue* | *DocumentValue* | *TextValue*

Item ::= *NodeValue* | *AtomicValue*

AtomicValue ::= *AtomicTypeValue* *TypeAnnotation*? (3.4)

AtomicTypeValue ::= *String* | *Boolean* | *Decimal* | *Float* | *Double*
 | *Duration* | *DateTime* | *Time* | *Date* | *GYearMonth*
 | *GYear* | *GMonthDay* | *GDay* | *GMonth* | *HexBinary*
 | *Base64Binary* | *AnyURI* | *QName* | *NOTATION*

TypeAnnotation ::= “of” “type” *TypeName*

ElementName ::= *QName* | *Wildcard* (3.5)

AttributeName ::= *QName* | *Wildcard* (3.6)

TypeName ::= *QName*

Em 3.1 o valor *error* não fazia parte da gramática. No entanto, como o próprio documento da semântica de XQuery aponta [DFFM02], o modelo de dados de XQuery faz referência a um valor de erro, mas ele não estava formalmente representado. Assim, ele foi incluído como produção para *Value*. Em 3.2, 3.3 e 3.4 *TypeAnnotation* não era opcional, no entanto, em algumas regras *TypeAnnotation* se comporta como tal.

Em 3.5 e 3.6 o *sort Wildcard* foi adicionado devido a inconsistências nas regras de análise de tipo estática para *node tests* (Seção 4.8.2). Tais regras aplicavam *wildcards* para *ElementName* e *AttributeName*, sem isto ser permitido, contudo, na gramática destes *sorts*. Tais regras fazem referência a *sorts* nas formas:

Prefix : *LocalPart*

★ : *LocalPart*

Prefix : ★

★

Uma destas formas é utilizada na regra a seguir, retirada da análise de tipos de *node tests* (Seção 4.8.2) :

$$\frac{LocalPart_1 = LocalPart_2}{statEnvs \vdash \text{element}, * : LocalPart_2 \text{ on element } * : LocalPart_1 \{Type\} : \text{element } * : LocalPart_2 \{Type\}}$$

Esta regra diz que o filtro (não-terminal *Filter* definido na Seção 4.8.2) **element**, **:LocalPart₂* sob o tipo **element** **:LocalPart₁ {Type}* tem como resultado da análise estática o tipo **element** **:LocalPart₂ {Type}* se a premissa *LocalPart₁ = LocalPart₂* for verdadeira. O problema que ocorre aqui é que o não-terminal **:LocalPart₁* (definido como *Wildcard*) que aparece na expressão de tipo **element** **:LocalPart₁ {Type}* não é compatível com a definição de nomes de elementos (*ElementName*), que permite somente *QNames*. A inclusão do não-terminal *Wildcard* como produção de *ElementName* se justifica desta forma. A modificação para *AttributeName* se justifica de forma semelhante.

3.4.2 Tipos

As seguintes produções de gramática apresentam a notação formal para tipos:

$$Type ::= ItemType \mid (Type \text{ OccurrenceIndicator}) \quad (3.7)$$

$$\begin{aligned} & \mid (Type \text{ "&" } Type) \mid (Type \text{ "," } Type) \\ & \mid (Type \text{ "|" } Type) \mid ("(" ")") \mid \text{"none"} \mid \text{"error"} \end{aligned} \quad (3.8)$$

$$\begin{aligned} ItemType ::= & \text{ElementType} \mid \text{AttributeType} \\ & \mid ("document" ("{" Type "}")) \\ & \mid \text{"text"} \mid \text{"untyped"} \\ & \mid \text{AtomicTypeName} \mid \text{"error"} \mid \text{"none"} \end{aligned} \quad (3.9)$$

$$\mid \text{AttributeAll} \mid \text{ElementContent} \quad (3.10)$$

$$\text{AtomicTypeName} ::= QName$$

Em 3.7 ao invés de *OccurrenceIndicator* havia um *sort* definido como *Occurrence*. Na sintaxe de XQuery *core* aparece definido um *sort OccurrenceIndicator*. Sendo que ambos possuíam

definição semelhante, o segundo foi retirado para evitar duplicação e ambigüidade e o primeiro foi modificado para contemplar as duas definições, sem prejuízo para a especificação. Em 3.8 *error* foi incluído por ser necessário em várias regras ao longo da especificação onde o erro de tipo aparece. Em 3.9 *error* e *none* não faziam parte da definição original de *ItemType*. Eles foram incluídos devido ao fato de que em *prime types* eles são referenciados utilizando *none*.

A referência 3.10 merece uma explicação mais detalhada. Os *sorts* *AttributeAll* e *ElementContent* (cuja sintaxe é mostrada no fim desta seção) não são citados como produção de nenhum outro *sort*, nem no sistema de tipos de XQuery, nem na gramática *core*. Este fato que faz com que eles não façam parte do sistema de tipos de XQuery, embora a definição apareça no sistema de tipos. Apesar disso, quando foi necessária a implementação das instruções auxiliares (Seção 3.4.5), esses dois *sorts* apareceram nas regras sendo usados como *ItemType*. Desta forma, a inclusão destes *sorts* como produções de *ItemType* se justifica.

As produções que completam o sistema de tipos de XQuery e não sofreram alterações são mostradas a seguir:

```

ElementType ::= "element" ElementName? Nillable? TypeSpecifier?
AttributeType ::= "attribute" AttributeName? TypeSpecifier?
Nillable ::= "nillable"
TypeSpecifier ::= TypeDerivation | TypeReference
TypeDerivation ::= Derivation? Mixed? "Type"?
TypeReference ::= "of" "type" TypeName
Derivation ::= ("restricts" TypeName) | ("extends" TypeName)
Mixed ::= "mixed"
Definition ::= ("define" "element" ElementName Substitution? Nillable? TypeSpecifier)
               | ("define" "attribute" AttributeName TypeSpecifier)
               | ("define" "type" TypeName TypeDerivation)
Substitution ::= "substitutes" "for" ElementName
SimpleType ::= AtomicTypeName | (SimpleType OccurrenceIndicator)
               | (SimpleType " | " SimpleType) | ("(" ")") | "none"
AttributeAll ::= AttributeType | (AttributeType "?")

```

$$\begin{aligned}
& | (AttributeAll \text{ "&" } AttributeAll) \mid (" " ") \\
ElementAll & ::= ElementType \mid (ElementType \text{ "?"}) \\
& | (ElementAll \text{ "&" } ElementAll) \mid (" " ") \mid \text{"none"} \\
ElementNoAll & ::= ElementType \mid \text{"text"} \mid (ElementType OccurrenceIndicator) \\
& | (ElementAll \text{ "," } ElementAll) \mid (ElementAll \text{ "|" } ElementAll) \\
& | (" " ") \\
ElementContent & ::= \text{"ElementContent"} \\
ElementBody & ::= AttributeAll ElementContent
\end{aligned}$$

3.4.3 Subtipos

A semântica de subtipos é denotada por sentenças da forma:

$$statEnv \vdash Type_1 <: Type_2$$

Esta notação indica que $Type_1$ é um subtipo de $Type_2$. Isto acontece se e somente se todas as instâncias no domínio de $Type_1$ estão também no domínio de $Type_2$ no ambiente $statEnv$.

3.4.4 Prime Types

Algumas expressões de XQuery, como por exemplo *for* e *unordered*, utilizam na definição de sua semântica estática a noção de *prime types*. *Prime types* representam uma escolha de itens, partindo de expressões que operam sobre seqüências de itens do mesmo tipo. A produção que descreve *prime types* é a seguinte:

$$PrimeType ::= ItemType \mid Type \mid (PrimeType \text{ "|" } PrimeType) \mid Type$$

O *sort Type* não fazia parte da definição na gramática de *PrimeType* e foi incluído porque as regras semânticas para *prime types* utilizam as produções de *Type*. Na implementação do protótipo isto foi verificado e incluído.

Para a aplicação de *prime types* foram utilizadas duas funções auxiliares da semântica definida em [DFFM02]. Estas funções são utilizadas em regras de inferência sem, no entanto, fazer parte da sintaxe de XQuery:

- a função `prime(Type)` que extrai todos os *ItemTypes* do tipo *Type* e os combina em uma escolha;
- a função `quantifier(Type)` que associa o possível número de *ItemTypes* em *Type* com os indicadores de ocorrência (*OccurrenceIndicator*) (`?`, `+`, `*`).

3.4.5 Instruções Auxiliares

O documento da semântica formal de XQuery [DFFM02] reserva uma seção para definição de instruções que especificam regras de relacionamento entre nomes, tipos, especificadores de tipos e valores. A semântica destas instruções é utilizada por algumas regras de verificação de tipos estática e pelas regras de avaliação dinâmica. Estas instruções foram implementadas no protótipo e estão descritas na Seção 4.7.1; elas são usadas na análise estática de tipos de *Axes* (Seção 4.8.2).

3.5 Semântica de Expressões

Esta seção, que define as expressões de XQuery, está organizada seguindo a mesma ordem em que a sua sintaxe foi abordada na Seção 2.2.

As regras de inferência usam o grupo do ambiente estático *StatEnv:s* que contém os ambientes construídos durante a verificação estática de tipos e que são usados tanto na verificação estática de tipos quanto na avaliação dinâmica. O grupo de ambiente estático mantém os seguintes ambientes:

- *namespace*: o ambiente de *namespace* mapeia os prefixos de *namespaces* (*NCName*) conforme suas declarações.
- *default_namespace*: o ambiente *padrão de namespace* mapeia *element* e *function* a *namespaces* de URIs.

- *typeDefn*: o ambiente de *definição de tipo* mapeia tipos de nomes (*TypeName*) a definições globais de tipo (*Definition*).
- *elemDecl*: o ambiente de *declaração de elementos* mapeia nomes de elementos (*ElementNames*) a sua declaração global de tipo de elemento (*Definition*).
- *attrDecl*: o o ambiente de *declaração de atributos* mapeia nomes de atributos (*AttributeNames*) a sua declaração global de tipo de atributo (*Definition*).
- *varType*: o ambiente *estático de tipo de variável* mapeia nomes de variáveis (*Variable \$ VarName*) para seus tipos estáticos (*Type*).
- *funcType*: o ambiente de *declaração de funções* armazena as *signatures* dos tipos estáticos das funções, além de mapear também o par (nome, aridade) para as funções. Isto é feito porque XQuery permite que várias funções tenham o mesmo nome, desde que possuam diferença no número de argumentos (sobrecarga). O mapeamento é feito através do nome da função a *define function QName (Type₁, ..., Type_n) returns Type*
- *base_uri*: o ambiente de *base uri* contém um único *namespace* (URI).

O uso de *dynEnvs* denota o grupo de ambientes dinâmicos, ou seja, aqueles ambientes construídos e utilizados durante a avaliação dinâmica. Os seguintes ambientes são dinâmicos:

- *funcDefn*: o ambiente de *função dinâmica* mapeia um nome de função a uma definição de função e assim como o ambiente *funcType*, este ambiente controla os pares (nome, aridade) para as funções.
- *varValue*: o ambiente de *valor dinâmico* mapeia um nome de variável (*QName*) para os valores atuais das variáveis (*Value*).

Para o protótipo foram construídos o ambiente estático e o ambiente dinâmico. A verificação estática de tipos foi implementada em sua maior parte, enquanto que somente uma pequena parte da avaliação dinâmica, necessária na definição de algumas produções auxiliares, foi implementada.

As seções seguintes descrevem as produções de gramática das expressões de XQuery após a normalização, portanto em XQuery *core*, bem como as regras que definem sua análise estática de tipos. Estas regras são transcritas conforme especificadas em [DFFM02], sem modificações. As correções e modificações feitas para implementação são apontadas e comentadas no Capítulo 4.

3.5.1 Expressões Primárias

As produções de gramática das expressões primárias de XQuery encontram-se na Seção 2.2.1. Nesta seção é exposta a semântica formal estática de variáveis, literais e chamadas de função.

Semântica Estática de Variáveis (*Variable*):

Gramática Core

$$\textit{Variable} ::= \$ \textit{QName}$$

Análise Estática de Tipo

$$\frac{\text{statEnvs.varType}(\textit{Variable}) = \textit{Type}}{\text{statEnvs} \vdash \textit{Variable} : \textit{Type}}$$

Semântica Estática de Literais (*Literal*):

Gramática Core

$$\textit{Literal} ::= \textit{NumericLiteral} \mid \textit{StringLiteral}$$

$$\textit{NumericLiteral} ::= \textit{IntegerLiteral} \mid \textit{DecimalLiteral} \mid \textit{DoubleLiteral}$$

Análise Estática de Tipo

O tipo de um literal inteiro é *xs:integer*.

$$\frac{}{\text{statEnvs} \vdash \textit{IntegerLiteral} : \text{xs:integer}}$$

As definições formais a seguir sobre literais decimais, *double* e *strings* são semelhantes às definições acima descritas para literais inteiros.

Análise Estática de Tipo

$$\frac{}{\text{statEnvs} \vdash \text{DecimalLiteral} : \text{xs} : \text{decimal}}$$

Análise Estática de Tipo

$$\frac{}{\text{statEnvs} \vdash \text{DoubleLiteral} : \text{xs} : \text{double}}$$

Análise Estática de Tipo

$$\frac{}{\text{statEnvs} \vdash \text{StringLiteral} : \text{xs} : \text{string}}$$

Semântica Estática de Chamadas de Função (*FunctionCall*):

Gramática Core

$$\text{FunctionCall} ::= \text{QName} "(" (\text{Expr} (", " \text{Expr}) *) ? ")"$$

Análise Estática de Tipo

Baseado no nome e número de argumentos, a *signature* da função é retornada do ambiente estático. Caso a função não esteja presente no ambiente, um erro estático é retornado. O tipo de cada argumento atual da função deve ser um subtipo do correspondente argumento formal.

$$\text{statEnvs} \vdash \text{QName} \Rightarrow \text{qname}$$

$$\text{statEnvs.funcType}(\text{qname}, n) = \text{qname}(\text{Type}_1, \dots, \text{Type}_n) \text{ returns } \text{Type}$$

$$\text{statEnvs} \vdash \text{Expr}_1 : \text{Type}'_1 \quad \text{Type}'_1 <: \text{Type}_1$$

...

$$\text{statEnvs} \vdash \text{Expr}_n : \text{Type}'_n \quad \text{Type}'_n <: \text{Type}_n$$

$$\text{statEnvs} \vdash \text{QName}(\text{Expr}_1, \dots, \text{Expr}_n) : \text{Type}$$

3.5.2 Expressões de Caminho

As produções de gramática das expressões de caminho de XQuery encontram-se na Seção 2.2.2. Nesta seção é exposta a semântica formal estática de expressões de caminho para *axes* e *node tests*.

Semântica Estática Expressões de Caminho (*PathExpr*):

Gramática Core

$$\begin{aligned} \textit{PathExpr} &::= \textit{RelativePathExpr} \\ \textit{RelativePathExpr} &::= \textit{StepExpr} \\ \textit{StepExpr} &::= \textit{ForwardStep} \mid \textit{ReverseStep} \mid \textit{PrimaryExpr} \\ \textit{ForwardStep} &::= \textit{ForwardAxis NodeTest} \\ \textit{ReverseStep} &::= \textit{ReverseAxis NodeTest} \end{aligned}$$

3.5.2.1 *Axes*

Semântica Estática Expressões de Caminho *Axes*:

Gramática Core

$$\begin{aligned} \textit{ForwardAxis} &::= \langle \textit{"child"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"descendant"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"attribute"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"self"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"descendant - or - self"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"following - sibling"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"following"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"namespace"} \textit{" ::"} \rangle \\ \textit{ReverseAxis} &::= \langle \textit{"parent"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"ancestor"} \textit{" ::"} \rangle \\ &\mid \langle \textit{"preceding - sibling"} \textit{" ::"} \rangle \end{aligned}$$

$$\begin{aligned}
& | \langle \text{"preceding"} \text{ " :: "} \rangle \\
& | \langle \text{"ancestor - or - self"} \text{ " :: "} \rangle
\end{aligned}$$

Algumas regras de produções de gramáticas auxiliares necessárias à análise de tipos de *axes* e *node tests* são abordadas a seguir. Elas não fazem parte da sintaxe de XQuery.

$$PrincipalNodeKind ::= \text{"element"} \mid \text{"attribute"} \mid \text{"namespace"}$$

$$Axis ::= ForwardAxis \mid ReverseAxis$$

$$Axis \text{ principal } PrincipalNodeKind$$

$$Filter ::= Axis \mid (PrincipalNodeKind \text{ } NodeTest)$$

Além destas produções, algumas declarações auxiliares são utilizadas. A primeira delas define o efeito de um filtro sobre um tipo na análise semântica e diz que a aplicação de um filtro *Filter* sobre o tipo *Type₁* resulta no tipo *Type₂*.

$$\mathbf{statEnvs} \vdash Filter \text{ on } Type_1 : Type_2$$

A segunda declaração utiliza um ambiente adicional de tipo *localTypeEnv*. Este ambiente faz o mapeamento de variáveis a tipos e é utilizado na análise estática de tipos de *axis* descendentes.

$$\mathbf{statEnvs}; localTypeEnv \vdash Filter \text{ on } Type_1 : Type_2$$

Análise Estática de Tipo

A semântica estática do par *Axis NodeTest* é obtida buscando no ambiente o tipo do nó de contexto *\$fs:dot* e aplicando um filtro a *Axis* e depois, com o resultado, a *NodeTest* com *PrincipalNodeKind*.

$$\mathbf{statEnvs}.varType(\$fs : dot) = Type_1$$

$$\mathbf{statEnvs} \vdash Axis \text{ on } Type_1 : Type_2$$

$$Axis \text{ principal } PrincipalNodeKind$$

$$\mathbf{statEnvs} \vdash PrincipalNodeKind, NodeTest \text{ on } Type_2 : Type_3$$

$$\mathbf{statEnvs} \vdash Axis \text{ } NodeTest : Type_3$$

As regras seguintes definem a semântica estática de filtros.

$$\frac{}{\text{statEnvs} \vdash \text{Filter on } () : ()}$$

$$\frac{}{\text{statEnvs} \vdash \text{Filter on none} : \text{none}}$$

$$\text{statEnvs} \vdash \text{Filter on } Type_1 : Type_3$$

$$\text{statEnvs} \vdash \text{Filter on } Type_2 : Type_4$$

$$\frac{}{\text{statEnvs} \vdash \text{Filter on } Type_1, Type_2 : Type_3, Type_4}$$

$$\text{statEnvs} \vdash \text{Filter on } Type_1 : Type_3$$

$$\text{statEnvs} \vdash \text{Filter on } Type_2 : Type_4$$

$$\frac{}{\text{statEnvs} \vdash \text{Filter on } Type_1 \mid Type_2 : Type_3 \mid Type_4}$$

$$\text{statEnvs} \vdash \text{Filter on } Type_1 : Type_3$$

$$\text{statEnvs} \vdash \text{Filter on } Type_2 : Type_4$$

$$\frac{}{\text{statEnvs} \vdash \text{Filter on } Type_1 \& Type_2 : Type_3 \& Type_4}$$

O tipo do nó principal (*PrincipalNodeKind*) para cada *axis* é definido pelas três regras a seguir.

$$\frac{}{\text{attribute} :: \text{principal attribute}}$$

$$\frac{}{\text{namespace} :: \text{principal namespace}}$$

$$Axis \neq \text{attribute} ::$$

$$Axis \neq \text{namespace} ::$$

$$Axis \text{ principal element}$$

A regra seguinte estabelece que o tipo do *axis self::* é o mesmo tipo do tipo do nó de contexto.

$$\frac{}{\text{statEnvs} \vdash \text{self} :: \text{on } NodeType : NodeType}$$

No caso de elementos, o tipo do *axis child::* é obtido pela extração dos tipos dos nós filhos.

$$\frac{}{\text{statEnvs} \vdash \text{child} :: \text{on element } qname \text{ AttrType, ChildType} : ChildType}$$

O tipo do *axis attribute::* é obtido pela extração dos tipos dos atributos.

$$\frac{}{\text{statEnvs} \vdash \text{attribute} :: \text{on element } qname \text{ AttrType, ChildType} : AttrType}$$

O tipo do *axis parent::* é um elemento ou um documento ou vazio.

$$\frac{}{\text{statEnvs} \vdash \text{parent} :: \text{on element} : (\text{element} \mid \text{document})?}$$

O tipo para *axis namespace::* é sempre vazio.

$$\frac{}{\text{statEnvs} \vdash \text{namespace} :: \text{on } NodeType : ()}$$

As regras seguintes verificam os tipos para *axis descendant::*, *descendant-or-self::*, *ancestor::* e *ancestor-or-self::*. Elas são implementadas através de aplicações recursivas de filtros. Algumas regras utilizam filtros com um ambiente local de tipos, que mantém um registro dos elementos visitados.

$$\frac{\text{statEnvs}; \{\} \vdash \text{descendant} :: \text{on } Type : Type'}{\text{statEnvs} \vdash \text{descendant} :: \text{on } Type : Type'}$$

As duas regras seguintes utilizam um ambiente auxiliar. Na primeira, o retorno é a união de todos os tipos dos nós visitados no ambiente local.

$$\frac{\begin{array}{l} \text{localTypeEnv}(qname) = Type \\ \text{localTypeEnv} = Type_1, Type_2, \dots \end{array}}{\text{statEnvs} \vdash \text{descendant} :: \text{on element } qname : (Type_1 \mid Type_2 \mid \dots)*}$$

Esta regra é utilizada quando o ambiente local não possui o tipo para um *qname* específico. Neste caso, é feita uma busca no ambiente `statEnvs.elemDecl` e uma inclusão do tipo encontrado no ambiente com uma regra auxiliar (`resolves to`) ao ambiente local.

$$\begin{array}{c}
\text{localTypeEnv}(qname) \text{ is a error} \\
\text{statEnvs.elemDecl}(qname) = \text{define element } qname \text{ TypeSpecifier} \\
\text{TypeSpecifier resolves to } \text{TypeName } Type \\
\hline
\text{statEnvs; localTypeEnv}[qname : Type] \vdash \text{descendant} :: \text{on } Type : Type' \\
\hline
\text{statEnvs localTypeEnv} \vdash \text{descendant} :: \text{on element } qname : Type'
\end{array}$$

$$\begin{array}{c}
\text{statEnvs} \vdash \text{child} :: \text{on } NodeType : Type_1 \\
\text{statEnvs} \vdash \text{descendant} :: \text{on } Type_1 : Type_2 \\
\hline
\text{statEnvs} \vdash \text{descendant} :: \text{on } NodeType : Type_1, Type_2
\end{array}$$

O tipo para o *axis descendant-or-self::* é obtido através da verificação de tipo do *axis self::* e do *axis descendant::*.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{self} :: \text{on } NodeType : Type_1 \\
\text{statEnvs} \vdash \text{descendant} :: \text{on } Type_1 : Type_2 \\
\hline
\text{statEnvs} \vdash \text{descendant - or - self} :: \text{on } NodeType : Type_1, Type_2
\end{array}$$

O tipo para o *axis ancestor::* é sempre o opcional do tipo `element`.

$$\begin{array}{c}
\hline
\text{statEnvs} \vdash \text{ancestor} :: \text{on } NodeType : \text{element*}
\end{array}$$

O tipo para o *axis ancestor-or-self::* é obtido através da verificação de tipo do *axis self::* e do *axis ancestor::*.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{self} :: \text{on } NodeType : Type_1 \\
\text{statEnvs} \vdash \text{ancestor} :: \text{on } Type_1 : Type_2 \\
\hline
\text{statEnvs} \vdash \text{ancestor - or - self} :: \text{on } NodeType : Type_1, Type_2
\end{array}$$

Em todos os outros casos, a aplicação de filtro resulta em um tipo vazio.

$$\begin{array}{c}
\hline
\text{statEnvs} \vdash \text{AxisName} :: \text{on } NodeType : ()
\end{array}$$

3.5.2.2 Node Tests

Semântica Estática Expressões de Caminho *Node Tests* (*NodeTest*):

Gramática Core

$$\begin{aligned}
 \text{NodeTest} &::= \text{KindTest} \mid \text{NameTest} \\
 \text{NameTest} &::= \text{QName} \mid \text{Wildcard} \\
 \text{Wildcard} &::= "*" \mid \langle \text{NCName} ":" "*" \rangle \mid \langle "*" ":" \text{NCName} \rangle \\
 \text{KindTest} &::= \text{ProcessingInstructionTest} \mid \text{CommentTest} \\
 &\quad \mid \text{TextTest} \mid \text{AnyKindTest} \\
 \text{ProcessingInstructionTest} &::= \text{"processing - instruction"} \text{"(" StringLiteral? ")} \\
 \text{CommentTest} &::= \text{"comment"} \text{"(" ")} \\
 \text{TextTest} &::= \text{"text"} \text{"(" ")} \\
 \text{AnyKindTest} &::= \text{"node"} \text{"(" ")}
 \end{aligned}$$

Análise Estática de Tipo

A verificação feita nesta primeira regra envolvendo elementos (**element**) separa $QName_2$ em suas partes constituintes e utiliza funções auxiliares (**xf:get-namespace-uri** e **xf:get-local-name**) para verificar a igualdade nos nomes. Como resultado, o tipo do filtro **element**, $QName_2$ sobre o tipo **element** $QName_1 \{Type\}$ é determinado.

$$\begin{array}{c}
 QName_2 = Prefix_2 : LocalPart_2 \\
 \text{xf : get - namespace - uri}(QName_1) = \text{statEnvs.varType}(Prefix_2) \\
 \text{xf : get - local - name}(QName_1) = LocalPart_2 \\
 \hline
 \text{statEnvs} \vdash \text{element, } QName_2 \text{ on element } QName_1 \{Type\} : \\
 \text{element } QName_1 \{Type\}
 \end{array}$$

Esta regra faz basicamente a verificação da igualdade das partes locais dos dois nomes pertencentes à regra. Caso a igualdade seja verdadeira, o tipo **element** $QName_2 \{Type\}$ é retornado.

$$\begin{array}{c}
 QName_2 = Prefix_2 : LocalPart_2 \\
 LocalPart_1 = LocalPart_2 \\
 \hline
 \text{statEnvs} \vdash \text{element, } QName_2 \text{ on element } * : LocalPart_1 \{Type\} : \\
 \text{element } QName_2 \{Type\}
 \end{array}$$

Após a verificação da igualdade das partes locais, esta regra utiliza o conteúdo do ambiente `statEnvs.namespace`.

$$\frac{QName_2 = Prefix_2 : LocalPart_2 \quad statEnvs.namespace(Prefix_1) = statEnvs.namespace(Prefix_2)}{statEnvs \vdash element, QName_2 \text{ on element } Prefix_1 : * \{Type\} : element Prefix_1 : LocalPart_2 \{Type\}}$$

O tipo do filtro `element QName2` sobre um tipo de elemento sem nome (sem *ElementName*) é `element QName2 {Type}`.

$$statEnvs \vdash element, QName_2 \text{ on element } \{Type\} : element QName_2 \{Type\}$$

Na verificação de filtros utilizando elementos da regra a seguir é utilizada uma função auxiliar que obtém a parte local de um *QName* (`xf:get-local-name`).

$$\frac{xf : get - local - name(QName_1) = LocalPart_2}{statEnvs \vdash element, * : LocalPart_2 \text{ on element } QName_1 \{Type\} : element QName_1 \{Type\}}$$

Esta regra utiliza a comparação das partes locais para fazer a verificação de tipo do filtro para *node test* sem prefixo.

$$\frac{LocalPart_1 = LocalPart_2}{statEnvs \vdash element, * : LocalPart_2 \text{ on element } * : LocalPart_1 \{Type\} : element * : LocalPart_2 \{Type\}}$$

Esta regra se aplica a filtros para *node tests* sem prefixo e nomes de elementos sem parte local.

$$statEnvs \vdash element, * : LocalPart_2 \text{ on element } Prefix_1 : * \{Type\} : element Prefix_1 : LocalPart_2 \{Type\}$$

O tipo do filtro `element *:LocalPart2` sobre um tipo de elemento sem nome (sem *ElementName*) é `element *:LocalPart2 {Type}`.

$$statEnvs \vdash element, * : LocalPart_2 \text{ on element } \{Type\} : element * : LocalPart_2 \{Type\}$$

Para determinar o tipo de filtro para *node test* sem parte local sobre tipos de elemento é utilizada a comparação entre o retorno da função `xf:get-namespace-uri` e o conteúdo do ambiente `statEnvs.namespace`.

$$\frac{\mathbf{xf} : \text{get} - \text{namespace} - \text{uri}(QName_1) = \text{statEnvs.namespace}(Prefix_2)}{\text{statEnvs} \vdash \text{element}, Prefix_2 : * \text{ on element } QName_1 \{Type\} : \text{element } QName_1 \{Type\}}$$

Esta regra se aplica a filtros para *node tests* sem parte local e nomes de elementos sem prefixo.

$$\frac{}{\text{statEnvs} \vdash \text{element}, Prefix_2 : * \text{ on element } * : LocalPart_1 \{Type\} : \text{element } Prefix_2 : LocalPart_1 \{Type\}}$$

Esta regra se aplica a filtros para *node tests* e nomes de elementos sem parte local e faz a comparação entre o conteúdo do ambiente `statEnvs.namespace` para o prefixo de cada um deles.

$$\frac{\text{statEnvs.namespace}(Prefix_1) = \text{statEnvs.namespace}(Prefix_2)}{\text{statEnvs} \vdash \text{element}, Prefix_2 : * \text{ on element } Prefix_1 : * \{Type\} : \text{element } Prefix_1 : * \{Type\}}$$

Esta regra se aplica a filtros para *node tests* sem parte local e sem nomes de elementos. O retorno é `element Prefix_2.* {Type}`.

$$\frac{}{\text{statEnvs} \vdash \text{element}, Prefix_2 : * \text{ on element } \{Type\} : \text{element } Prefix_2 : * \{Type\}}$$

Esta última regra relacionada a elementos retorna o mesmo tipo sobre o qual o filtro é aplicado.

$$\frac{}{\text{statEnvs} \vdash \text{element}, * \text{ on element } prefix : local \{Type\} : \text{element } prefix : local \{Type\}}$$

As regras seguintes são semelhantes às comentadas acima com a diferença de serem aplicadas a atributos.

$$\begin{aligned} QName_2 &= Prefix_2 : LocalPart_2 \\ \mathbf{xf} : \text{get} - \text{namespace} - \text{uri}(QName_1) &= \text{statEnvs.varType}(Prefix_2) \end{aligned}$$

$$\frac{\text{xf} : \text{get} - \text{local} - \text{name}(QName_1) = LocalPart_2}{\text{statEnvs} \vdash \text{attribute}, QName_2 \text{ on attribute } QName_1 \{Type\} : \text{attribute } QName_1 \{Type\}}$$

$$\frac{QName_2 = Prefix_2 : LocalPart_2 \quad LocalPart_1 = LocalPart_2}{\text{statEnvs} \vdash \text{attribute}, QName_2 \text{ on attribute } * \cdot LocalPart_1 \{Type\} : \text{attribute } QName_2 \{Type\}}$$

$$\frac{QName_2 = Prefix_2 : LocalPart_2 \quad \text{statEnvs.namespace}(Prefix_1) = \text{statEnvs.namespace}(Prefix_2)}{\text{statEnvs} \vdash \text{attribute}, QName_2 \text{ on attribute } Prefix_1 : * \{Type\} : \text{attribute } Prefix_1 : LocalPart_2 \{Type\}}$$

$$\text{statEnvs} \vdash \text{attribute}, QName_2 \text{ on attribute } \{Type\} : \text{attribute } QName_2 \{Type\}$$

$$\frac{\text{xf} : \text{get} - \text{local} - \text{name}(QName_1) = LocalPart_2}{\text{statEnvs} \vdash \text{attribute}, * : LocalPart_2 \text{ on attribute } QName_1 \{Type\} : \text{attribute } QName_1 \{Type\}}$$

$$\frac{LocalPart_1 = LocalPart_2}{\text{statEnvs} \vdash \text{attribute}, * : LocalPart_2 \text{ on attribute } * : LocalPart_1 \{Type\} : \text{attribute } * : LocalPart_2 \{Type\}}$$

$$\text{statEnvs} \vdash \text{attribute}, * : LocalPart_2 \text{ on attribute } Prefix_1 : * \{Type\} : \text{attribute } Prefix_1 : LocalPart_2 \{Type\}$$

$$\text{statEnvs} \vdash \text{attribute}, * : LocalPart_2 \text{ on attribute } \{Type\} : \text{attribute } * : LocalPart_2 \{Type\}$$

$$xf : \text{get} - \text{namespace} - \text{uri}(QName_1) = \text{statEnvs.namespace}(Prefix_2)$$

$$\text{statEnvs} \vdash \text{attribute}, Prefix_2 : * \text{ on attribute } QName_1 \{Type\} : \\ \text{attribute } QName_1 \{Type\}$$

$$\text{statEnvs} \vdash \text{attribute}, Prefix_2 : * \text{ on attribute } * : LocalPart_1 \{Type\} : \\ \text{attribute } Prefix_2 : LocalPart_1 \{Type\}$$

$$\text{statEnvs.namespace}(Prefix_1) = \text{statEnvs.namespace}(Prefix_2) \\ \text{statEnvs} \vdash \text{attribute}, Prefix_2 : * \text{ on attribute } Prefix_1 : * \{Type\} : \\ \text{attribute } Prefix_1 : * \{Type\}$$

$$\text{statEnvs} \vdash \text{attribute}, Prefix_2 : * \text{ on attribute } \{Type\} : \\ \text{attribute } Prefix_2 : * \{Type\}$$

$$\text{statEnvs} \vdash \text{attribute}, * \text{ on attribute } prefix : local \{Type\} : \\ \text{attribute } prefix : local \{Type\}$$

O tipo de um filtro com *node test* instrução de processamento sobre o tipo instrução de processamento tem como resultado na análise estática o tipo **processing-instruction**.

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{processing} - \text{instruction}() \\ \text{on processing} - \text{instruction} : \text{processing} - \text{instruction}$$

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{processing} - \text{instruction}(String) \\ \text{on processing} - \text{instruction} : \text{processing} - \text{instruction}$$

O tipo de um filtro com *node test* comentário sobre o tipo comentário tem como resultado na análise estática o tipo **comment**.

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{comment}() \text{ on comment} : \text{comment}$$

O tipo de um filtro com *node test* texto sobre o tipo texto tem como resultado na análise estática o tipo **text**.

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{text}() \text{ on } \text{text} : \text{text}$$

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{node}() \text{ on } \text{NodeType} : \text{NodeType}$$

No caso das regras acima não serem aplicadas, então a sequência vazia é retornada.

$$\text{statEnvs} \vdash \text{PrincipalNodeKind}, \text{node}() \text{ on } \text{NodeType} : ()$$

3.5.3 Expressões de Sequência

As produções de gramática das expressões de sequência de XQuery encontram-se na Seção 2.2.3.

Nesta seção é exposta a semântica formal estática de tais expressões.

Semântica Estática de Expressões de Sequência (*ExprSequence*):

Gramática Core

$$\text{ExprSequence} ::= \text{Expr} (", " \text{Expr})^*$$

Análise Estática de Tipo

O tipo sobre uma sequência de expressões é a sequência sobre os tipos das expressões individuais.

$$\frac{\text{statEnvs} \vdash \text{Expr}_1 : \text{Type}_1 \ \cdots \ \text{statEnvs} \vdash \text{Expr}_n : \text{Type}_n}{\text{statEnvs} \vdash \text{Expr}_1, \cdots, \text{Expr}_n : \text{Type}_1, \cdots, \text{Type}_n}$$

3.5.4 Expressões Aritméticas

As produções de gramática das expressões aritméticas de XQuery encontram-se na Seção 2.2.4. Nesta seção é exposta a semântica formal estática de tais expressões que especificam os tipos resultantes dos operadores aritméticos quando aplicados a tipos específicos.

Semântica Estática de Expressões Aritméticas (*ValueExpr*):

Gramática Core

$$ValueExpr ::= ValidateExpr \mid CastExpr \mid Constructor \mid PathExpr$$

Análise Estática de Tipo

As regras a seguir especificam os tipos resultantes para os operadores aritméticos quando aplicadas a tipos numéricos específicos. A operação entre duas expressões resulta no tipo decimal quando os tipos das duas expressões são subtipos do tipo decimal.

$$\frac{\begin{array}{l} statEnvs \vdash Expr_1 : Type_1 \\ Type_1 <: xs : decimal \\ statEnvs \vdash Expr_2 : Type_2 \\ Type_2 <: xs : decimal \end{array}}{statEnvs \vdash op : operation(Expr_1, Expr_2) : xs : decimal}$$

A operação entre duas expressões resulta no tipo *float* quando os tipos das duas expressões são subtipos do tipo *float*.

$$\frac{\begin{array}{l} statEnvs \vdash Expr_1 : Type_1 \\ Type_1 <: xs : float \\ statEnvs \vdash Expr_2 : Type_2 \\ Type_2 <: xs : float \end{array}}{statEnvs \vdash op : operation(Expr_1, Expr_2) : xs : float}$$

A operação entre duas expressões resulta no tipo *double* quando os tipos das duas expressões são subtipos do tipo *double*.

$$statEnvs \vdash Expr_1 : Type_1$$

$$\begin{array}{c}
Type_1 <: xs : double \\
statEnvs \vdash Expr_2 : Type_2 \\
\hline
Type_2 <: xs : double \\
\hline
statEnvs \vdash op : operation(Expr_1, Expr_2) : xs : double
\end{array}$$

A operação entre duas expressões resulta no tipo inteiro quando os tipos das duas expressões são subtipos do tipo inteiro.

$$\begin{array}{c}
statEnvs \vdash Expr_1 : Type_1 \\
Type_1 <: xs : integer \\
statEnvs \vdash Expr_2 : Type_2 \\
Type_2 <: xs : integer \\
\hline
statEnvs \vdash op : operation(Expr_1, Expr_2) : xs : integer
\end{array}$$

No caso de operadores unários as regras são semelhantes.

A operação entre uma expressão resulta no tipo decimal quando o tipo da expressão é subtipo do tipo decimal.

$$\begin{array}{c}
statEnvs \vdash Expr_1 : Type_1 \\
Type_1 <: xs : decimal \\
\hline
statEnvs \vdash op : operation(Expr_1) : xs : decimal
\end{array}$$

A operação entre uma expressão resulta no tipo *float* quando o tipo da expressão é subtipo do tipo *float*.

$$\begin{array}{c}
statEnvs \vdash Expr_1 : Type_1 \\
Type_1 <: xs : float \\
\hline
statEnvs \vdash op : operation(Expr_1) : xs : float
\end{array}$$

A operação entre uma expressão resulta no tipo *double* quando o tipo da expressão é subtipo do tipo *double*.

$$\begin{array}{c}
statEnvs \vdash Expr_1 : Type_1 \\
Type_1 <: xs : double \\
\hline
statEnvs \vdash op : operation(Expr_1) : xs : double
\end{array}$$

A operação entre uma expressão resulta no tipo inteiro quando o tipo da expressão é subtipo do tipo inteiro.

$$statEnvs \vdash Expr_1 : Type_1$$

$$\frac{Type_1 <: xs : integer}{statEnvs \vdash op : operation(Expr_1) : xs : integer}$$

3.5.5 Construtores

As produções de gramática dos construtores de XQuery encontram-se na Seção 2.2.7. Nesta seção é exposta a semântica formal estática de tais construtores.

Semântica Estática de Construtores (*Constructor*):

Gramática Core

$$\begin{aligned} Constructor &::= XmlComment \mid XmlProcessingInstruction \\ &\quad \mid ComputedDocumentConstructor \\ &\quad \mid ComputedElementConstructor \\ &\quad \mid ComputedAttributeConstructor \\ EnclosedExpr &::= (\mid \{ \}) ExprSequence \{ \} \\ ComputedElementConstructor &::= (("element" QName \{ \}) \\ &\quad \mid (("element" \{ \}) Expr \{ \} \{ \}) ExprSequence? \{ \}) \\ ComputedAttributeConstructor &::= (("attribute" QName \{ \}) \\ &\quad \mid (("attribute" \{ \}) Expr \{ \} \{ \}) ExprSequence? \{ \}) \\ ComputedDocumentConstructor &::= "document" \{ \} ExprSequence? \{ \} \\ XmlProcessingInstruction &::= "<?" PITarget Char * ">" \\ XmlComment &::= "<!-- Char * -->" \end{aligned}$$

Análise Estática de Tipo

Na regra a seguir, o tipo para o construtor elemento é obtido através da aplicação da função **expand** ao nome do elemento.

$$\frac{statEnvs \vdash expand(QName) = qname}{statEnvs \vdash element QName \{ ExprSequence \} : element qname \{ xs : anyType \}}$$

Na regra a seguir, o tipo para o construtor elemento com expressão é obtido através da verificação do tipo da expressão *Expr*.

$$\frac{\text{statEnvs} \vdash \text{Expr} : \text{xs} : \text{QName}}{\text{statEnvs} \vdash \text{element} \{ \text{Expr} \} \{ \text{ExprSequence} \} : \text{element} \{ \text{xs} : \text{anyType} \}}$$

Na regra a seguir, o tipo para o construtor atributo com expressão é obtido através da verificação do tipo da expressão *Expr*.

$$\frac{\text{statEnvs} \vdash \text{Expr} : \text{xs} : \text{QName}}{\text{statEnvs} \vdash \text{attribute} \{ \text{Expr} \} \{ \text{ExprSequence} \} : \text{attribute} \{ \text{xs} : \text{anySimpleType} \}}$$

Na regra a seguir, o tipo para o construtor atributo é obtido através da aplicação da função **expand** ao nome do elemento.

$$\frac{\text{statEnvs} \vdash \text{expand}(\text{QName}) = \text{qname}}{\text{statEnvs} \vdash \text{attribute } \text{QName} \{ \text{ExprSequence} \} : \text{attribute } \text{qname} \{ \text{xs} : \text{anySimpleType} \}}$$

3.5.6 Expressões FLWR

As produções de gramática das expressões FLWR encontram-se na Seção 2.2.8. Nesta seção é exposta a semântica formal da expressão *for* e da expressão *let*, já que a cláusula *where* é normalizada para uma expressão condicional *IfExpr*, que será vista na Seção 3.5.8.

Semântica da Expressão For (*ForExpr*):

Gramática Core

$$\begin{aligned} \text{ForExpr} &::= (\text{ForClause } \text{"return"}) * \text{TypeswitchExpr} \\ \text{ForClause} &::= \text{"for"} \text{TypeDeclaration? } \text{"\$"} \text{VarName } \text{"in"} \text{Expr} \\ \text{TypeDeclaration} &::= \text{SequenceType} \end{aligned}$$

Análise Estática de Tipo

Inicialmente *Type₁* é inferido; então o tipo *prime* de *Type₁* é determinado. A variável do ambiente estático é estendida com *Variable₁* do tipo *prime* de *Type₁* e então *Type₂* de *Expr₂* é inferido. Devido a expressão *for* iterar sobre o resultado de *Expr₁*, o tipo final da iteração

é $Type_2$ multiplicado ao possível número de itens em $Type_1$ (um, ?, * ou +). Este número determinado pelo quantificador auxiliar de tipo de função $quantifier(Type_1)$.

$$\frac{\text{statEnvs} \vdash Expr_1 : Type \quad \text{statEnvs}[\text{varType}(Variable_1 : \text{prime}(Type_1))] \vdash Expr_2 : Type_2}{\text{statEnvs} \vdash \text{for } Variable_1 \text{ in } Expr_1 \text{ return } Expr_2 : Type_2.\text{quantifier}(Type_1)}$$

No caso de uma sequência de tipos estar presente ($SequenceType$) o tipo retornado pela normalização é verificado quanto a relação de subtipo com o tipo da expressão ($Expr_1$) *for*.

$$\frac{\begin{array}{l} \text{statEnvs} \vdash Expr_1 : Type \\ Type_0 = [SequenceType]_{sequencetype} \\ Type_1 <: Type_0 \\ \text{statEnvs}[\text{varType}(Variable_1 : \text{prime}(Type_1))] \vdash Expr_2 : Type_2 \end{array}}{\text{statEnvs} \vdash \text{for } SequenceType \ Variable_1 \text{ in } Expr_1 \text{ return } Expr_2 : Type_2.\text{quantifier}(Type_1)}$$

Semântica da Expressão Let ($LetExpr$):

Gramática Core

$$\begin{aligned} LetExpr &::= (LetClause \text{ "return" }) * TypeswitchExpr \\ ForClause &::= \text{ "let" } TypeDeclaration? \text{ "$" } VarName \text{ " := " } Expr \\ TypeDeclaration &::= SequenceType \end{aligned}$$

Análise Estática de Tipo

A expressão *let* estende o ambiente $statEnvs$ com $Variable_1$ do tipo $Type_1$ inferido de $Expr_1$ e infere o tipo de $Expr_2$ no ambiente estendido para produzir o tipo resultado $Type_2$.

$$\frac{\text{statEnvs} \vdash Expr_1 : Type_1 \quad \text{statEnvs}[\text{varType}(Variable_1 : Type_1)] \vdash Expr_2 : Type_2}{\text{statEnvs} \vdash \text{let } Variable_1 := Expr_1 \text{ return } Expr_2 : Type_2}$$

No caso de uma sequência de tipos estar presente ($SequenceType$) o tipo retornado pela normalização é verificado quanto a relação de subtipo com o tipo da expressão ($Expr_1$) *for*.

$$\text{statEnvs} \vdash Expr_1 : Type$$

$$\begin{array}{c}
Type_0 = [SequenceType]_{sequence_type} \\
Type_1 <: Type_0 \\
\hline
statEnvs[varType(Variable_1 : (Type_1))] \vdash Expr_2 : Type_2 \\
\hline
statEnvs \vdash let Variable_1 in Expr_1 return Expr_2 : Type_2
\end{array}$$

3.5.7 Expressões de Ordem

As produções de gramática de expressões de ordem de XQuery encontram-se na Seção 2.2.9. Nesta seção é exposta a semântica formal estática de tais expressões.

Semântica Estática de Expressões de Ordem (*SortExpr*):

Gramática Core

$$\begin{array}{l}
SortExpr ::= UnorderedExpr ("stable"? "sort" "by" "(" SortSpecList ")") \\
SortSpecList ::= Expr SortModifier ("," SortSpecList)? \\
SortModifier ::= ("ascending" | "descending") \\
\quad ("empty" "greatest") | ("empty" "least") \\
UnorderedExpr ::= "unordered"? ForExpr
\end{array}$$

Análise Estática de Tipo

Nesta regra são utilizadas as funções auxiliares de tipo **prime**(*Type*) e **quantifier**(*Type*), que estão definidas na Seção 3.4.4. O tipo da expressão *Expr* é determinado e as funções auxiliares são aplicadas a ele.

$$\frac{statEnvs \vdash Expr_1 : Type_1}{statEnvs \vdash unordered Expr : prime(Type_1).quantifier(Type_1)}$$

3.5.8 Expressões Condicionais

As produções de gramática das expressões condicionais encontram-se na Seção 2.2.10. Nesta seção é exposta a semântica formal estática de tais expressões.

Semântica de Expressões Condicionais (*IfExpr*):

Gramática Core

$$IfExpr ::= ("if" "(" Expr ")" "then" Expr "else") * ValueExpr$$

Análise Estática de Tipo

O tipo de uma expressão condicional é a disjunção dos tipos de cada uma das duas expressões que compõem a expressão condicional ($Expr_2$ e $Expr_3$) se a primeira expressão ($Expr_1$) for do tipo booleano.

$$\frac{\begin{array}{l} statEnvs \vdash Expr_1 : xs : \text{boolean} \\ statEnvs \vdash Expr_2 : Type_2 \\ statEnvs \vdash Expr_3 : Type_3 \end{array}}{statEnvs \vdash \text{if } (Expr_1) \text{ then } Expr_2 \text{ else } Expr_3 : (Type_2 \mid Type_3)}$$

3.5.9 Expressões sobre Sequência de Tipos

As produções de gramática das expressões sobre sequência de tipos encontram-se na Seção 2.2.12. Nesta seção é exposta a semântica formal estática de expressões sobre sequência de tipos *cast*. A semântica para *typeswitch* não é descrita aqui por não ter sido implementada no protótipo.

Semântica de Expressões sobre Sequência de Tipos (*CastExpr*):

Gramática Core

$$CastExpr ::= "cast" "as" SequenceType ParenthesizedExpr$$

Análise Estática de Tipo

Para a análise de expressões de *cast* é necessária a consulta a uma tabela com os tipos onde o *cast* é permitido (Y), não é permitido (N) ou pode ser feito em alguns casos (M). Se a tabela de *cast* indica que o *cast* não é permitido, o sistema aponta um erro de tipo estático. Caso contrário, as regras abaixo retornam a semântica estática destas expressões.

Se o tipo estático da expressão $Type_1$ pode ser a sequência vazia, então o tipo estático do *cast* é o opcional de $Type_2$.

$$\begin{array}{c}
\text{statEnvs} \vdash Expr : Type_1 \\
\text{quantifier}(Type_1) = ? \text{ or } \text{quantifier}(Type_1) = * \\
Type_2 = [SequenceType_2]_{sequencetype} \\
\hline
Type_1 \text{ cast allowed } Type_2 \Rightarrow Y \text{ or } Type_1 \text{ cast allowed } Type_2 \Rightarrow M \\
\hline
\text{statEnvs} \vdash \text{cast as } SequenceType_2 (Expr) : Type_2?
\end{array}$$

Se o tipo estático da expressão $Type_1$ não pode ser a sequência vazia, então o tipo estático do *cast* é $Type_2$.

$$\begin{array}{c}
\text{statEnvs} \vdash Expr : Type_1 \\
\text{quantifier}(Type_1) \neq ? \text{ quantifier}(Type_1) \neq * \\
Type_2 = [SequenceType_2]_{sequencetype} \\
\hline
Type_1 \text{ cast allowed } Type_2 \Rightarrow Y \text{ or } Type_1 \text{ cast allowed } Type_2 \Rightarrow M \\
\hline
\text{statEnvs} \vdash \text{cast as } SequenceType_2 (Expr) : Type_2
\end{array}$$

3.5.10 Expressões de Validação

As produções de gramática das expressões de validação de XQuery encontram-se na Seção 2.2.13. Nesta seção é abordada somente a gramática *core* de tais expressões, já que não há regras estáticas definidas para elas.

Gramática Core

$$ValidateExpr ::= \text{"validate"} SchemaContext? \{ " Expr " \}$$

3.5.11 Observações Finais

Para a maioria das expressões de XQuery estão definidas regras estáticas que avaliam completamente ou em partes as expressões. Tais regras foram descritas ao longo deste capítulo. No entanto, conforme o documento de semântica formal de XQuery [DFFM02], algumas expressões de XQuery não possuem correspondente em XQuery *core* e não possuem regras de

análise estática definidas. São elas: expressões de comparação, expressões lógicas e expressões quantificadas. Algumas destas expressões são normalizadas para outras expressões *core*.

Capítulo 4

Verificação Estática de Tipos de XQuery

Este capítulo aborda os detalhes de implementação da verificação de tipos de XQuery *core* no ambiente ASF+SDF. Para o desenvolvimento do protótipo foram usadas como referência as especificações de [DFFM02].

4.1 Construção do ambiente do protótipo

O protótipo construído é uma combinação da sintaxe e da semântica da linguagem XQuery *core*. Além da linguagem propriamente dita, o protótipo também foi construído através da implementação de definições e notações auxiliares fornecidas pelo próprio documento de semântica formal de XQuery. Estas complementam as regras e auxiliam em verificações extras que devem ser feitas antes de se fornecer a resposta se uma expressão é corretamente tipada. O meta-ambiente ASF+SDF, a partir destas especificações, produziu como resultado um interpretador para a linguagem.

A especificação formal completa consiste de dezesseis módulos que serão descritos nas seções subsequentes. O conjunto dos módulos pode ser dividido em sete subconjuntos:

1. Básico: este subconjunto contém um módulo:

- *Layout*.
2. Sintáticos: mostram a sintaxe de XQuery e é especificada em dois módulos:
- *Terminals*
 - *XQueryCoreSyntax*.
3. Sistema de Tipos: mostram a sintaxe e semântica que envolvem o sistema de tipos de XQuery e conta com os seguintes módulos:
- *TypeSystem*
 - *SubTypes*
 - *PrimeTypes*
 - *TypeInteger*
 - *TypeBoolean*.
4. Ambientes: mostram os ambientes que registram a informação computada durante a análise estática de tipos ou avaliação dinâmica e possui os seguintes módulos:
- *StatEnvironment*
 - *DynEnvironment*.
5. Funções e Operadores: este subconjunto possui dois módulos, baseados nas especificações de [MMRW02]:
- *FunctionsOnQNames*
 - *CastingTable*.
6. Auxiliares: este subconjunto é formado pelos módulos que contém notações e especificações extras, que não fazem parte da sintaxe formal de XQuery, mas auxiliam na verificação de tipos e avaliação dinâmica. Apresenta também o módulo que contém a única normalização implementada, para sequência de tipos:
- *AuxiliaryJudgments*

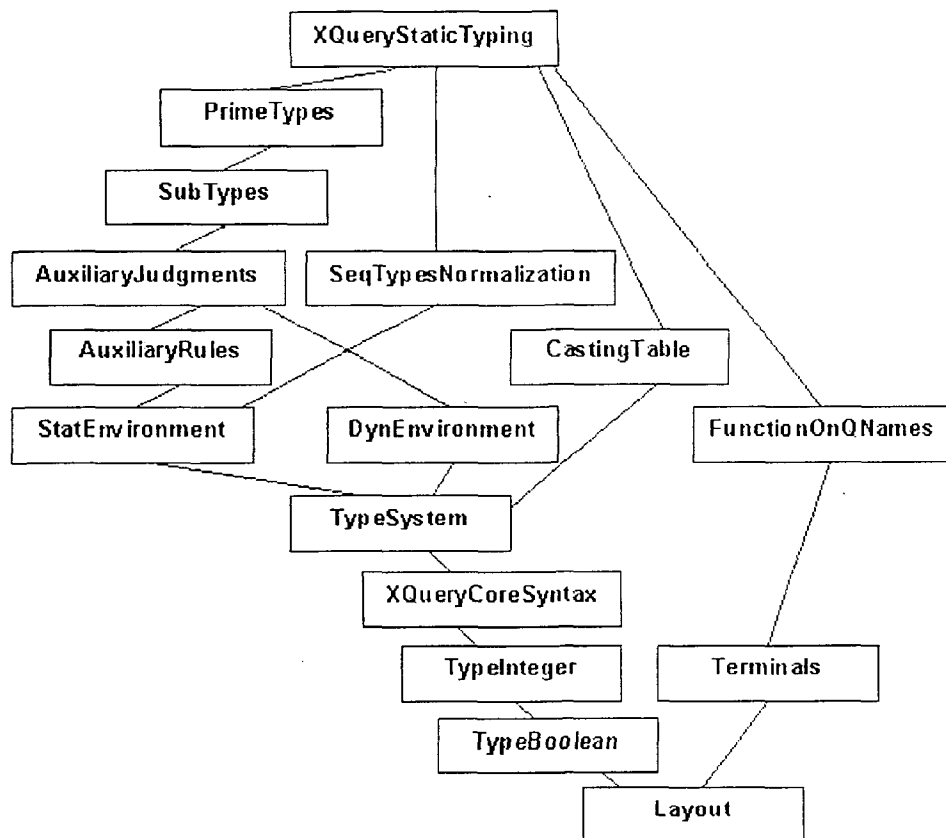


Figura 4.1: Grafo do projeto.

- *AuxiliaryRules*
- *SeqTypesNormalization*.

7. Verificação de Tipos: este subconjunto possui o módulo principal do protótipo:

- *XQueryStaticTyping*.

A figura 4.1 mostra o grafo do projeto. O grafo mostra cada módulo do projeto e suas dependências com outros módulos. A descrição mais detalhada de cada módulo será abordada nas seções seguintes.

4.2 Módulo Básico

Esta seção aborda a implementação do arquivo de *layout*, que é utilizado em toda especificação ASF+SDF.

Este módulo define a sintaxe que é padrão do meta-ambiente. Comentários são definidos como `%%`, finalizando com um novo caractere de linha (`~[\n]*[\n]`). São definidos como caracteres padrão ainda o espaço, a tabulação (`[\t]`) e o caractere de nova linha (`[\n]`). No meta-ambiente ASF+SDF está implementado no arquivo `Layout.sdf`, sendo mostrado no Apêndice A.1.

4.3 Módulos Sintáticos

Esta seção aborda a implementação da estrutura sintática da linguagem XQuery *core*, com os terminais e não-terminais.

4.3.1 *TerminalsSyntax*

Este módulo contém a gramática dos terminais de XQuery *core*. No meta-ambiente ASF+SDF está implementada no arquivo `TerminalsSyntax.sdf`, sendo mostrada no Apêndice A.2.

4.3.2 *XQueryCoreSyntax*

Este módulo contém a gramática dos não-terminais pertencentes a XQuery *core*. No meta-ambiente ASF+SDF está implementada no arquivo `XQueryCoreSyntax.sdf`. Um trecho de sintaxe de XQuery *core* é mostrado a seguir, na notação BNF.

$$\begin{aligned} \textit{PrimaryExp} ::= & \textit{Literal} \mid \textit{FunctionCall} \mid (\textit{"\$"} \textit{VarName}) \\ & \mid \textit{ParenthesizedExpr} \end{aligned}$$

Estas regras de produções de gramática expressam que expressões primárias (*PrimaryExpr*) são formadas por literais (*Literal*), por chamadas de função (*FunctionCall*), por variáveis

(`"$"` *VarName*) ou por expressões parentetizadas (*ParenthesizedExpr*). No meta-ambiente ASF+SDF estas regras são traduzidas para:

```
Literal | FunctionCall | (" $" VarName) | ParenthesizedExpr -> PrimaryExpr
```

As correções foram feitas na gramática devido a inconsistências, observadas durante a implementação. Algumas delas são comentadas a seguir.

Na especificação do não-terminal *SortExpr* foi acrescentado o símbolo `?`, tornando parte da produção opcional. Isto justifica-se pela análise da gramática da linguagem: se esta parte não for opcional, ocorre um *loop* sintático e nenhuma *query* seria aceita pela linguagem.

Analisando a gramática de XQuery *core* percebe-se que aparecem produções para o não-terminal *LetExpr* sem, contudo, este ser apontado em nenhum outro lugar como produção de outro não-terminal. Isto faz com que ele não faça parte da linguagem XQuery *core*. No entanto, ele faz parte da linguagem XQuery como produção da gramática número nove, conforme [DFFM02]:

```
[9 (XQuery)]  UnorderedExpr ::= "unordered"? FLWRExpr
[10 (XQuery)] FLWRExpr      ::= ((ForClause | LetClause)+
                                WhereClause? "return" ) * QuantifiedExpr
```

Em XQuery *core* o *sort* *UnorderedExpr* estava definido como a seguir:

```
[7 (Core)] UnorderedExpr ::= "unordered"? ForExpr
```

Note que a expressão que está sendo definida não inclui a cláusula *let* que faz parte de XQuery *core*. Isto faz concluir que pode ter havido um erro na normalização de expressões *unordered*, que transforma expressões XQuery em expressões XQuery *core*. Como correção, foi acrescentado o *sort* *LetExpr* como produção de *UnorderedExpr*:

```
[7 (Core)] UnorderedExpr ::= "unordered"? (ForExpr | LetExpr)+
```

Outra correção feita foi para o *sort* *ItemType*. O sistema de tipos de XQuery (Seção 3.4) também possui um *sort* definido como *ItemType*. Ao longo da implementação da especificação

percebeu-se que o uso deste não-terminal é feito sem mencionar se a referência é sobre o *ItemType* da gramática ou o do sistema de tipos. Assim, assumiu-se que *ItemType* da gramática é o mesmo que *ItemType* do sistema de tipos, unindo suas produções. Para manter a modularidade, está declarado em *XQueryCoreSyntax* e suas produções estão em *XQueryCoreSyntax* e também em *TypeSystem*. Para não gerar ambigüidade, foram retirados *text* e *untyped* da produção de *ItemType*, porque estes símbolos já estavam definidos em *TypeSystem*.

Além destas, algumas pequenas modificações de eliminação de caracteres extras foram feitas, baseadas em exemplos de XQuery *core*.

O código da implementação deste módulo pode ser encontrado no Apêndice A.3.

4.4 Módulos do Sistema de Tipos

Esta seção aborda a implementação da sintaxe do sistema de tipos de XQuery. Inclui as definições do sistema de tipos propriamente dito e seus afins: subtipos e *prime types*. Finalmente, definem-se a sintaxe e semântica para tipos inteiros e booleanos.

4.4.1 *TypeSystem*

Os comentários sobre as modificações e correções no sistema de tipos de XQuery são feitos nas Seções 3.4.1 e 3.4.2. No meta-ambiente ASF+SDF a sintaxe para o sistema de tipos está implementada no arquivo *TypeSystem.sdf* e é mostrada no Apêndice A.4.

4.4.2 *SubTypes*

A sintaxe e a semântica de subtipos de XQuery são comentadas na Seção 3.4.3. No meta-ambiente ASF+SDF estão implementadas nos arquivos *SubTypes.sdf* e *SubTypes.asf*, mostrados no Apêndice A.5.

4.4.3 *PrimeTypes*

A sintaxe e a semântica de *prime types* foram comentadas na Seção 3.4.4. No meta-ambiente ASF+SDF estão implementadas nos arquivos `PrimeTypeSyntax.sdf`, mostrados no Apêndice A.6.

4.4.4 *TypeInteger*

A sintaxe e semântica de tipos inteiros para uso em XQuery *core* são baseadas na especificação *pico*, que é mostrada como exemplo no manual do meta-ambiente ASF+SDF [BK03]. No meta-ambiente está implementada nos arquivos `TypeInteger.sdf` e `TypeInteger.asf`, mostrados no Apêndice A.7.

4.4.5 *TypeBoolean*

A sintaxe e semântica de tipos booleanos para uso em XQuery *core* são baseadas na especificação *pico*, que é mostrada como exemplo no manual do meta-ambiente ASF+SDF [BK03]. No meta-ambiente está implementada nos arquivos `TypeBoolean.sdf` e `TypeBoolean.asf`, mostrados no Apêndice A.8.

4.5 Módulos de Ambiente

Esta seção descreve a implementação da base para o ambiente estático de XQuery *core*. O ambiente dinâmico também está implementado, embora seja usado em poucas partes do protótipo.

4.5.1 *StatEnvironment*

As regras de inferência estáticas usam o grupo de ambiente `StatEnv` que contém as informações do ambiente construído durante a verificação estática de tipos. Estas informações são usadas durante a verificação estática de tipos e também na avaliação dinâmica. As especificações do grupo do ambiente `StatEnv` estão definidas teoricamente no início da Seção 3.5. No meta-ambiente

ASF+SDF o ambiente estático está implementado nos arquivos `StatEnvironment.sdf` e `StatEnvironment.asf`.

O ambiente `StatEnv` é composto por oito listas de pares que basicamente fazem mapeamentos de chaves (*Key*) a determinados conteúdos (*Contents*). Estas listas representam os grupos do ambiente e são: `namespace`, `default-namespace`, `type-defn`, `elem-decl`, `attr-decl`, `var-type`, `func-type` e `base-uri`. Cada componente de `StatEnv` faz um mapeamento diferente que é armazenado no ambiente. O trecho de código abaixo estabelece a ordem dos componentes no ambiente, onde `LP` é uma variável para o não-terminal *ListOfPairs*.

```
[stat-1] namespace(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP1
[stat-2] default-namespace(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP2
[stat-3] type-defn(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP3
[stat-4] elem-decl(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP4
[stat-5] attr-decl(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP5
[stat-6] var-type(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP6
[stat-7] func-type(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP7
[stat-8] base-uri(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP8
```

O componente de ambiente `namespace`, por exemplo, tem seu mapeamento armazenado na primeira lista de pares, o `default-namespace` na segunda lista e assim por diante.

O mapeamento pode estar no ambiente inicialmente ou ser inserido nele explicitamente através da função `insert`. Para cada componente do ambiente está definida uma instância da função `insert`. Abaixo, segue um trecho de código que implementa a função `insert` para os dois primeiros componentes do ambiente: `namespace (NS)` e `default-namespace (DNS)`.

```
[insert-1]
    insert P in NS of [Ps], LP2, LP3, LP4, LP5, LP6, LP7, LP8
        = [P Ps], LP2, LP3, LP4, LP5, LP6, LP7, LP8
[insert-2]
    insert P in DNS of LP1, [Ps], LP3, LP4, LP5, LP6, LP7, LP8
        = LP1, [P Ps], LP3, LP4, LP5, LP6, LP7, LP8
```

A função `insert` insere um par `P` no ambiente de acordo com o componente desejado (`NS`, `DNS`, ...). Como resultado tem-se um ambiente sendo construído.

Quando é necessário recuperar o conteúdo de um componente do ambiente, uma função de busca nas listas de pares é ativada (`lookup`). O código a seguir, no meta-ambiente ASF+SDF, mostra a implementação da função `lookup`.

```
[look-a] lookup(K, []) = error
[look-b]
    lookup(K, [(K --> C) Ps]) = C
[look-c]
    K != K1,
    lookup(K, [Ps]) = C2
    =====
    lookup(K, [(K1 --> C1) Ps]) = C2
```

A regra `[look-a]` é utilizada caso a busca seja mal sucedida, isto, é, caso a chave (variável K do não-terminal *Key*) não seja encontrada no ambiente. As regras `[look-b]` e `[look-c]` percorrem os pares comparando a chave a ser encontrada com a chave de cada par que está no ambiente. A regra `[look-b]` é utilizada no caso da chave K procurada ser igual a chave do primeiro par e, assim, retorna o conteúdo deste par (variável C do não-terminal *Contents*). Caso a chave procurada não seja igual a chave do primeiro par (regra `[look-c]`), uma busca nos pares seguintes é iniciada (variável Ps representando a lista de pares restantes).

A última função definida no arquivo de sintaxe é a `expand`, que expande o *sort* `QName` utilizando o conteúdo do ambiente estático `namespace`.

O código completo da implementação deste módulo é mostrado no Apêndice A.9.

4.5.2 *DynEnvironment*

`DynEnv` denota o grupo de ambiente construído e usado somente durante a avaliação dinâmica. As especificações do grupo do ambiente `dynEnv` estão definidas teoricamente no início da Seção 3.5. No meta-ambiente ASF+SDF o ambiente dinâmico está implementado nos arquivos `DynEnvironment.sdf` e `DynEnvironment.asf`.

O ambiente `DynEnv` é, em sua estrutura, bastante semelhante ao `StatEnv`. Ele é composto por duas listas de pares que basicamente fazem mapeamentos de chaves (*KeyDyn*) a deter-

minados conteúdos (`ContentsDyn`). Estas listas representam os grupos do ambiente e são: `func-defn` e `var-value`. Cada componente de `DynEnv` faz um mapeamento diferente que é armazenado no ambiente. Este mapeamento pode estar no ambiente inicialmente ou ser inserido nele explicitamente através da função `insert`. Quando é necessário recuperar o conteúdo do ambiente, uma função de busca nas listas de pares é ativada (`lookupDyn`).

O código completo da implementação deste módulo é mostrado no Apêndice A.10

4.6 Módulos de Funções e Operadores

O módulo de Funções e Operadores se caracteriza por possuir implementação de funções e notações auxiliares baseadas fora do documento de semântica formal de XQuery. Esta seção apresenta comentários sobre a implementação de dois módulos que foram construídos utilizando material extra apontado pelo documento de semântica formal. *XQuery 1.0 and XPath 2.0 Functions and Operators* [MMRW02] é também um trabalho em desenvolvimento e define funções e operadores básicos para serem usados em XQuery e outros padrões relacionados a XML.

4.6.1 *FunctionsOnQNames*

A análise estática de tipos de *node tests* (Seção 4.8.2) faz referências a duas funções extras necessárias para complementar verificação de tipos. São elas `xf:get-namespace-uri` e `xs:get-local-name`. Em [MMRW02] encontram-se suas definições:

- `xf:get-namespace-uri(QName)`: retorna a URI de `QName`;
- `xs:get-local-name(QName)`: retorna uma *string* representando a parte local de `QName`.

O código mostrado a seguir corresponde à implementação da semântica para a função auxiliar `xf:get-namespace-uri(QName)`.

```
[names-1]
  QN = NCN1:NCN2
  =====
```

```
xf:get-namespace-uri(QN) = NCN1
```

A função é chamada com um *QName* (QN) como parâmetro e retorna o prefixo (NCN1) do mesmo. A função `xs:get-local-name(QName)` possui implementação e funcionamento semelhantes, retornando a parte local (NCN2) de QN.

No meta-ambiente ASF+SDF estas funções estão implementadas nos arquivos `FunctionsOnQNames.sdf` e `FunctionsOnQNames.asf`, mostrados no Apêndice A.11.

4.6.2 *CastingTable*

Na análise estática de tipos de expressões *cast* (Seção 4.8.9) é necessário saber se o *cast* de um determinado tipo é permitido para outro. Para obter esta informação foi preciso implementar uma tabela de *cast* que indica se uma conversão é permitida ou não. A semântica das funções de *cast* estão descritas através de tabelas e esquemas em [MMRW02]. Para este projeto foi implementada a tabela que faz o *cast* de tipos primitivos para tipos primitivos. As respostas para o *cast* são:

- Y: indica que a conversão dos valores dos tipos mencionados é permitida;
- N: indica que a conversão dos valores dos tipos mencionados não é permitida;
- M: indica que a conversão dos valores dos tipos mencionados pode ser permitida, sujeita a restrições.

Um trecho de implementação da tabela é mostrado a seguir.

```
[1] T = T1
=====
T cast allowed T1 = Y
[2] xs:string cast allowed xs:float   = M
[20] xs:float cast allowed xs:string  = Y
[21] xs:float cast allowed xs:double  = Y
[269] xs:hexBinary cast allowed xs:time = N
```

A regra [1] diz que o *cast* entre tipos iguais é permitido. A regra [2] diz que o *cast* entre o tipo *string* e o tipo *float* pode ser permitido. As regras [20] e [21] dizem que o *cast* do tipo *float* para os tipos *string* e *double* é permitido. A regra [269] diz que não é permitido o *cast* entre o tipo *hexBinary* para o tipo *time*.

No meta-ambiente ASF+SDF a tabela completa de tipos primitivos para tipos primitivos está implementada nos arquivos `CastingTable.sdf` e `CastingTable.asf`, mostrados no Apêndice A.12.

4.7 Módulos de Funções Auxiliares

O módulo de Funções Auxiliares se caracteriza por possuir implementação de funções e notações auxiliares baseadas e fornecidas pelo documento de semântica formal de XQuery. Esta seção aborda a implementação de três módulos que serão descritos a seguir.

4.7.1 *AuxiliaryJudgments*

Na análise estática de tipos de *axes* (Seção 4.8.2) aparece uma chamada para verificar se um determinado especificador de tipo (`TypeSpecifier`) possui um relacionamento de *resolução* com um nome de tipo (`TypeName`), como no trecho de código a seguir:

...
TypeSpecifier resolves to TypeName Type
 ...

Tal premissa exige a implementação de funções auxiliares, que são fornecidas no documento de semântica formal de XQuery e fazem verificações que respondem com verdadeiro (*true*) ou falso (*false*) a assertivas. A regra acima ficou definida no protótipo sintaticamente como a seguir.

```
(" TypeSpecifier "resolves" "to" TypeName "{" Type "}" " ," StatEnvs ") -> Boolean
```

De forma semelhante acontece com as demais assertivas.

Algumas regras utilizam o ambiente dinâmico de XQuery, justificando sua especificação no projeto. Foram implementadas quarenta e cinco regras estáticas das oitenta e cinco regras

existentes e todas as quatro regras dinâmicas. No meta-ambiente ASF+SDF estão implementadas nos arquivos `AuxiliaryJudgments.sdf` e `AuxiliaryJudgments.asf` e são mostradas no Apêndice A.13.

4.7.2 *AuxiliaryRules*

Na especificação da análise estática de tipos de *axis* se fez necessário a criação e manutenção de um ambiente adicional (`localTypeEnv`), que mapeia variáveis a tipos e é utilizado nas regras de verificação de tipos de *axis* descendentes. Este ambiente adicional está implementado neste módulo.

No meta-ambiente ASF+SDF estas regras estão implementadas nos arquivos `AuxiliaryRules.sdf` e `AuxiliaryRules.asf` e são mostrados no Apêndice A.14.

4.7.3 *SeqTypesNormalization*

Esta seção descreve a implementação da normalização de *SequenceTypes*, utilizada na análise estática de tipos de expressões FLWR, conforme seção 4.8.6. A notação fornecida pelo documento de semântica formal [DFFM02] para normalização de *SequenceType* é:

$$\begin{array}{c} [SequenceType]_{sequencetype} \\ == \\ Type \end{array}$$

Esta notação especifica que *SequenceType* é mapeado para *Type*. No meta-ambiente, a sintaxe para a normalização de *SequenceType* ficou definida como a seguir.

```
module SeqTypesNormalization
imports StatEnvironment
exports
  sorts ListSeqType ListTypeOr
  context-free syntax
  ("normSeqT" "(" SequenceType ")") -> Type {prefer}
  ("normSeqT-list" "(" ListSeqType ")") -> ListTypeOr {prefer}
```

```
"[" {SequenceType ","}* "]"          -> ListSeqType
"[" {Type "|"}* "]"                  -> ListTypeOr
```

A função `normSeqT` (e similarmente `normSeqT-list`) recebe como parâmetro uma seqüência de tipos (ou uma lista de seqüência de tipos) retornando um tipo (ou uma lista disjuntiva de tipos). Semânticamente, a normalização fica como mostrado no trecho de código a seguir.

```
[seqt-1]
    normSeqT(IT1) = T
    =====
    normSeqT(IT1 OI1) = T OI1

[seqt-4] normSeqT(attribute SchemaT) = attribute SchemaT

[seqt-5] normSeqT(element QN) = element QN
```

A regra [seqt-1] estabelece que indicadores de ocorrências (variável `OI1` do não-terminal *OccurrenceIndicator*) não sofrem modificação quando normalizados com *SequenceTypes*. A normalização de um *ItemType* (`IT1`) seguido de um indicador de ocorrência (`OI1`) tem como resultado o tipo obtido da normalização do *ItemType* seguido do indicador de ocorrência.

As regras [seqt-4] e [seqt-5] o mapeamento direto de *ItemTypes* para o sistema de tipos.

No meta-ambiente ASF+SDF a normalização para *SequenceType* está implementada nos arquivos `SeqTypesNormalization.sdf` e `SeqTypesNormalization.asf` e é mostrada no Apêndice A.15.

4.8 Módulo de Verificação de Tipos

Esta seção descreve a implementação do módulo principal do protótipo. Ele contém a descrição da sintaxe necessária à verificação de tipos das expressões de XQuery *core*, bem como sua semântica. Este módulo contém a verificação de tipos de:

1. expressões primárias;
2. expressões de caminho;

3. expressões de sequência;
4. expressões aritméticas;
5. construtores;
6. expressões FLWR;
7. expressões relacionadas a ordem;
8. expressões condicionais;
9. expressões sobre sequência de tipos (de *cast*).

Oitenta e uma regras, das oitenta e seis existentes, foram implementadas no meta-ambiente ASF+SDF. As regras para *typeswitch* não foram implementadas. No meta-ambiente ASF+SDF, as regras estão implementadas nos arquivos `XQueryStaticTyping.sdf` e `XQueryStaticTyping.asf`.

A função `typecheck` é a base para a verificação de tipos das expressões de XQuery *core* implementada neste trabalho. A seguir é mostrada a definição desta função para alguns casos, implementada no módulo *XQueryStaticTyping*, no arquivo de sintaxe.

```
"typecheck" "(" QueryBody "," StatEnvs ")"      -> Type {avoid}
"typecheck" "(" ExprSequence "," StatEnvs ")"    -> ListType
"typecheck" "(" ParenthesizedExpr "," StatEnvs ")" -> ListType
"typecheck" "(" ArithmeticExpr "," StatEnvs ")"  -> Type {prefer}
```

A função `typecheck` recebe como parâmetros uma expressão (ou mais de uma, em alguns casos) e os valores que compõem o ambiente estático. Baseada nestas entradas, esta função é utilizada nas regras de verificação de tipos ao longo do protótipo, tendo como saída o tipo (ou tipos) da expressão fornecida.

As subseções subsequentes mostram partes do código deste módulo e comentários sobre elas. O código completo de *XQueryStaticTyping* é mostrado no Apêndice A.16.

4.8.1 Expressões Primárias

A sintaxe de expressões primárias de XQuery *core* é mostrada na Seção 2.2.1. A seguir, é mostrada a verificação estática de tipos de expressões primárias, implementada no meta-ambiente ASF+SDF.

```
[lite-1] typecheck(IntegerLit, statEnvs) = xs:integer
[lite-2] typecheck(DecimalLit, statEnvs) = xs:decimal
[lite-3] typecheck(DoubleLit, statEnvs) = xs:double
[lite-4] typecheck(StringLit, statEnvs) = xs:string

[var]
    lookup($ Var, var-type(statEnvs)) = T
    =====
    typecheck($ Var, statEnvs) = T

[func]
    FuncCall = QN(E1, Es),
    lookup((QN), func-type(statEnvs)) =
        define function QN(T1, Ts) returns T,
    typecheck-list([E1, Es], statEnvs) = [T2, Ts2],
    subtype-list([T2, Ts2], [T1, Ts], statEnvs) = true
    =====
    typecheck(FuncCall, statEnvs) = T

[default-func] typecheck(FuncCall, statEnvs) = static error
```

Os axiomas [lite-1] a [lite-4] mostram a semântica estática para expressões literais. O tipo de um literal inteiro é `xs:integer`, de um literal decimal é `xs:decimal` e de forma semelhante para literal *double* e literal *string*.

A regra [var] diz que o tipo de uma variável é o seu tipo armazenado no ambiente de `statEnvs var-type`.

A regra [func] estabelece que o tipo de uma chamada de função deve ser primeiramente obtido através de uma busca no ambiente de `statEnvs func-type`. Se esta busca não for bem sucedida, um erro é retornado e a verificação de tipo de `FuncCall` retorna um erro estático. Caso a função esteja no ambiente, é feita uma verificação de tipos nos parâmetros da função e

uma comparação para subtipos.

4.8.2 Expressões de Caminho

A sintaxe para expressões de caminho de XQuery *core* é mostrada na Seção 3.5.2. A verificação de tipos de expressões de caminho envolve a tipagem de *axes* e *node tests*. Algumas regras fazem parte e auxiliam tanto na verificação de tipos de *axes* quanto de *node tests*. Estas regras utilizam alguns *sort* auxiliares:

- **PrincipalNodeKind**: no código representado pela variável PNK. Associado aos valores *element*, *attribute* e *namespace*;
- **Axis**: no código representado pela variável Ax. Este *sort* combina os *sorts* *ForwardAxis* e *ReverseAxis*;
- **Filter**: no código representado pela variável F. Este *sort* denota ou um *Axis* ou um *PrincipalNodeKind* com *NodeTest*.

A seguir, é mostrada e comentada em partes a verificação estática de tipos de expressões de caminho, implementada no meta-ambiente ASF+SDF.

```
[path-1] attribute:: principal attribute = true
[path-2] namespace:: principal namespace = true
[path-3]
    Ax != attribute::,
    Ax != namespace::
    =====
    Ax principal element = true
[default-path] Ax principal PNK = false
```

As regras [path-1], [path-2] e [path-3] acima, utilizam uma sentença auxiliar que especifica o *PrincipalNodeKind* para cada *Axis*. A regra [default-path] especifica que o padrão para definição do *axis* principal é resultar em *false*.

Observe a seguinte regra fornecida pelo documento de semântica formal [DFFM02] para *Axis NodeTest*:

$$\begin{array}{c}
\text{statEnvs.varType}(\$fs : \text{dot}) = \text{Type}_1 \\
\text{statEnvs} \vdash \text{Axis on } \text{Type}_1 : \text{Type}_2 \\
\text{Axis principal } \text{PrincipalNodeKind} \\
\hline
\text{statEnvs} \vdash \text{PrincipalNodeKind, NodeTest on } \text{Type}_2 : \text{Type}_3 \\
\text{statEnvs} \vdash \text{Axis NodeTest} : \text{Type}_3
\end{array}$$

Esta regra é traduzida no meta-ambiente ASF+SDF para as regras [axnode-1a] a [axnode-1c] mostradas a seguir.

[axnode-1a]

```

lookup($fs:dot, var-type(statEnvs)) = T1,
typecheck(Ax on T1, statEnvs) = T2,
Ax principal element = true,
typecheck((element, NTest) on T2, statEnvs) = T3
=====
typecheck(Ax NTest, statEnvs) = T3

```

[axnode-1b]

```

lookup($fs:dot, var-type(statEnvs)) = T1,
typecheck(Ax on T1, statEnvs) = T2,
Ax principal attribute = true,
typecheck((attribute, NTest) on T2, statEnvs) = T3
=====
typecheck(Ax NTest, statEnvs) = T3

```

[axnode-1c]

```

lookup($fs:dot, var-type(statEnvs)) = T1,
typecheck(Ax on T1, statEnvs) = T2,
Ax principal namespace = true,
typecheck((namespace, NTest) on T2, statEnvs) = T3
=====
typecheck(Ax NTest, statEnvs) = T3

```

A tradução se dá em três regras por ser necessária a substituição dos valores para o *sort* *PrincipalNodeKind*: *element*, *attribute* e *namespace*. Esta substituição ocorreu devido a rigidez sintática do meta-ambiente na validação das regras. Estas regras verificam o tipo do par *Ax*is *NodeTest* primeiramente buscando no ambiente (através do componente *var-type*) o tipo do nó de contexto \$ *fs:dot*. O tipo retornado pela aplicação de um filtro a *Ax*is (variável *Ax*) é usado sobre *PrincipalNodeKind* ((*element*, *NTest*), por exemplo), determinando o tipo para *Ax*is *NodeTest* (*Ax* *NTest*).

As regras seguintes definem a semântica estática para filtros (variável *F* do não-terminal *Filter*).

```
[axnode-2] typecheck(F on (), statEnvs) = ()
[axnode-3] typecheck(F on none, statEnvs) = none
[axnode-4]
    typecheck(F on T1, statEnvs) = T3,
    typecheck(F on T2, statEnvs) = T4
    =====
    typecheck(F on T1 , T2, statEnvs) = (T3 , T4)
[axnode-5]
    typecheck(F on T1, statEnvs) = T3,
    typecheck(F on T2, statEnvs) = T4
    =====
    typecheck(F on T1 | T2, statEnvs) = T3 | T4
[axnode-6]
    typecheck(F on T1, statEnvs) = T3,
    typecheck(F on T2, statEnvs) = T4
    =====
    typecheck(F on T1 & T2, statEnvs) = T3 & T4
```

A regra [axnode-2] especifica que a aplicação de um filtro *F* a um tipo vazio é um tipo vazio. De forma semelhante, a regra [axnode-3] resulta em um tipo *none* se o filtro é aplicado a um tipo *none*. As regras [axnode-4], [axnode-5] e [axnode-6] definem a análise de tipos para filtros aplicados a lista de tipos (*T*₁ , *T*₂), disjunção de tipos (*T*₁ | *T*₂) e conjunção de tipos (*T*₁ & *T*₂). Para cada caso, filtros são aplicados aos tipos envolvidos, originando tipos

resultantes (T_3 e T_4) que são combinados para gerar a saída da regras.

As regras seguintes [axes-1] a [axes-14] definem a semântica estática de `Filter` aplicado a *axes*. Nestas regras de inferência as variáveis `NodeT`, `AttrT` e `ChildT` são do tipo `Type`, assim como `T`, `T1`, e semelhantes. `NodeT`, `AttrT` e `ChildT` foram utilizadas, como sugestão do documento de semântica formal de XQuery [DFFM02], para melhorar a legibilidade das regras.

```
[axes-1] typecheck(self:: on NodeT, statEnvs) = NodeT
[axes-2] typecheck(child:: on element QN{AttrT, ChildT}, statEnvs) = ChildT
[axes-3] typecheck(attribute:: on element QN{AttrT, ChildT}, statEnvs) = AttrT
[axes-4] typecheck(parent:: on element, statEnvs) = (element | document) | none
[axes-5] typecheck(namespace:: on NodeT, statEnvs) = ()
```

A regra [axes-1] determina que o filtro `self::` resulta no mesmo tipo ao qual ele é aplicado. As regras [axes-2] e [axes-3] aplicam, respectivamente, os filtros `child::` e `attribute` a um tipo `element QN{AttrT, ChildT}`. O resultado desta verificação de tipos é a extração dos tipos filhos e atributos, respectivamente, do tipo ao qual a regra é aplicada. O tipo para o *axis* `parent::` aplicado a um elemento, conforme a regra [axes-4], é um elemento, um documento ou nenhum tipo. Finalmente, o tipo para o *axis* `namespace::` é sempre vazio.

As regras seguintes verificam os tipos para *axis* do tipo `descendant::`, `descendant-or-self::`, `ancestor::` e `ancestor-or-self::`.

```
[axes-6]
    localTypeEnv = [],
    typecheck(descendant:: on T, statEnvs, localTypeEnv) = T1
    =====
    typecheck(descendant:: on T, statEnvs, localTypeEnv) = T1
[axes-7]
    lookup-local(QN, localTypeEnv) = T,
    T != error,
    extract-types-local(localTypeEnv) = [Ts],
    modify [Ts] to [] = AltT
    =====
    typecheck(descendant:: on element QN, statEnvs, localTypeEnv) = AltT
```

As regras [axes-6] e [axes-7] foram modificadas das suas formas originais no documento de semântica formal de XQuery. Estas regras, juntamente com a [axes-8] (abaixo), computam o tipo estático de *axis* descendentes utilizando o ambiente adicional `localTypeEnv`. Contudo, embora as regras [axes-6] e [axes-7] estivessem em suas premissas utilizando tal ambiente, este não era passado a elas. A correção feita foi a adição de `localTypeEnv` ao lado esquerdo da conclusão a fim de torná-lo utilizável pelas regras. A regra original, retirada da semântica de XQuery, correspondente à regra [axes-7] é a seguinte:

$$\frac{\begin{array}{l} \text{localTypeEnv}(qname) = Type \\ \text{localTypeEnv} = Type_1, Type_2, \dots \end{array}}{\text{statEnvs} \vdash \text{descendant} :: \text{on element } qname : (Type_1 \mid Type_2 \mid \dots)*}$$

O retorno é a disjunção de todos os tipos dos nós visitados no ambiente local. Para se chegar a isso, inicialmente, a função de busca no ambiente local (Seção 4.7.2) retorna um tipo diferente de `error`, isto é, o ambiente local possui o valor procurado. Duas funções adicionais são necessárias: `extract-types-local` e `modify` (Seção 4.7.2). A primeira faz uma busca no ambiente local, retornando uma lista dos tipos encontrados. A segunda função transforma uma lista de tipos em uma disjunção de tipos. Isto é necessário porque o resultado da regra apresenta os tipos ligados ao conectivo `|`.

Com a seguinte regra acontece algo semelhante.

[axes-8]

```
lookup-local(QN, localTypeEnv) = error,
lookup((QN), elem-decl(statEnvs)) = Def,
Def = define element QN TSpec,
(TSpec resolves to TName {T}, statEnvs) = true,
insert-local (QN --> T) of localTypeEnv = localTypeEnv1,
typecheck(descendant:: on T, statEnvs, localTypeEnv1) = AltT
=====
typecheck(descendant:: on element QN, statEnvs, localTypeEnv) = AltT
```

A regra [axes-8] é o resultado da tradução da seguinte da semântica formal de XQuery para o meta-ambiente:

localTypeEnv(*qname*) is a error

$$\begin{array}{c}
\text{statEnvs.elemDecl}(qname) = \text{define element } qname \text{ TypeSpecifier} \\
\text{TypeSpecifier resolves to } \text{TypeName Type} \\
\hline
\text{statEnvs; localTypeEnv}[qname : \text{Type}] \vdash \text{descendant} :: \text{on Type : Type}' \\
\text{statEnvs localTypeEnv} \vdash \text{descendant} :: \text{on element } qname : \text{Type}'
\end{array}$$

Esta regra é utilizada quando o ambiente local não possui o tipo para um *qname* específico. Neste caso, a função de busca no ambiente local (Seção 4.7.2) retorna `error`. Em seguida, é feita uma busca pelo elemento no componente do ambiente estático `statEnvs.elemDecl` utilizando a função `lookup`, bem como uma verificação utilizando a sentença auxiliar (Seção 4.7.1) `resolves to` é feita. Antes da obtenção final de tipos, o nome do elemento encontrado deve ser incluído no ambiente local através da função `insert-local` (Seção 4.7.2) gerando um ambiente local atualizado. Finalmente, é feita uma nova chamada para verificação de tipos para o *axis* `descendant::` para obtenção do tipo resultante da regra.

As quatro últimas regras que aplicam filtro a *axes* são mostradas a seguir.

[axes-9a]

```

typecheck(child:: on NodeT, statEnvs) = T1,
typecheck(descendant:: on T1, statEnvs) = T2
=====
typecheck(descendant:: on NodeT, statEnvs) = (T1 , T2)

```

[axes-9b]

```

typecheck(self:: on NodeT, statEnvs) = T1,
typecheck(descendant:: on T1, statEnvs) = T2
=====
typecheck(descendant-or-self:: on NodeT, statEnvs) = (T1 , T2)

```

[axes-10a] typecheck(ancestor:: on NodeT, statEnvs) = element | ()

[axes-10b]

```

typecheck(self:: on NodeT, statEnvs) = T1,
typecheck(ancestor:: on T1, statEnvs) = T2
=====
typecheck(ancestor-or-self:: on NodeT, statEnvs) = (T1 , T2)

```

Em todos os outros casos, a aplicação de filtro resulta em um tipo vazio, conforme a regra original da semântica de XQuery.

$$\text{statEnvs} \vdash \text{AxisName} :: \text{on NodeType} : ()$$

A regra acima é ambígua para o meta-ambiente ASF+SDF então, os casos restantes foram especificados um a um no protótipo, conforme a seguir.

[axes-11] `typecheck(following-sibling:: on NodeT, statEnvs) = ()`

[axes-12] `typecheck(following:: on NodeT, statEnvs) = ()`

[axes-13] `typecheck(preceding-sibling:: on NodeT, statEnvs) = ()`

[axes-14] `typecheck(preceding:: on NodeT, statEnvs) = ()`

As regras seguintes [node-1] a [node-31] definem a semântica estática de `Filter` aplicado a *node tests*. Para que estas regras fossem implementadas e funcionassem corretamente, uma atualização de inclusão do *sort Wildcard* como produção de `ElementName` e `AttributeName` foi feita no sistema de tipos de XQuery. Esta atualização está descrita em 3.4.1.

[node-1]

```

xf:get-namespace-uri(QN1) = NCN,
lookup(Pre2, namespace(statEnvs)) = NCN,
xf:get-local-name(QN1) = NCN1,
=====
typecheck((element, Pre2:NCN1) on element QN1 {T}, statEnvs) =
                                                    (element QN1 {T})

```

A regra acima, originalmente na semântica de XQuery, estava definida como:

$$\begin{array}{c}
 QName_2 = Prefix_2 : LocalPart_2 \\
 \text{xf : get - namespace - uri}(QName_1) = \text{statEnvs.varType}(Prefix_2) \\
 \text{xf : get - local - name}(QName_1) = LocalPart_2 \\
 \hline
 \text{statEnvs} \vdash \text{element, } QName_2 \text{ on element } QName_1 \{Type\} : \\
 \text{element } QName_1 \{Type\}
 \end{array}$$

A regra estabelecia uma comparação de igualdade entre o resultado da função `xf:get-namespace-uri` e o resultado de uma busca no ambiente de `statEnvs var-type` (`lookup`). Esta igualdade nunca seria verdadeira, porque o ambiente `var-type` mapeia nomes de variáveis a seus

tipos, o que não pode ser igual a uma URI, que é o que a função `xf:get-namespace-uri` retorna. O ambiente correto a ser utilizado neste caso é o namespace de `statEnvs`. Tal alteração foi efetuada no protótipo.

As regras seguintes testam *node tests* sobre elementos e atributos, buscando sempre resultar no tipo mais específico possível. As regras [node-1] a [node-13] se aplicam a elementos e [node-14] a [node-26] se aplicam a atributos.

A regra a seguir faz basicamente a verificação da igualdade das partes locais dos dois nomes pertencentes à regra. Caso a igualdade seja verdadeira, o tipo `element Pre2:NCN2 {T}` é retornado.

[node-2]

```
NCN1 = NCN2,
=====
typecheck((element, Pre2:NCN2) on element *:NCN1 {T}, statEnvs) =
                                                    (element Pre2:NCN2 {T})
```

A regra [node-3] utiliza o conteúdo do ambiente `statEnvs.namespace`, obtido através da função `lookup`. O resultado da aplicação da função a cada termo (`NCN1` e `Pre2`) é comparado e o tipo da expressão é retornado.

[node-3]

```
lookup(NCN1, namespace(statEnvs)) = lookup(Pre2, namespace(statEnvs))
=====
typecheck((element, Pre2:Loc2) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:Loc2 {T})
```

Na regra a seguir, o tipo do filtro `element QN2` sobre um tipo de elemento sem nome (sem *ElementName*) é `element QN2 {T}`.

[node-4]

```
typecheck((element, QN2) on element {T}, statEnvs) = (element QN2 {T})
```

Na verificação de filtros utilizando elementos da regra a seguir é utilizada uma função auxiliar que obtém a parte local de um *QName* `QN1` (`xf:get-local-name`).

[node-5]

```
xf:get-local-name(QN1) = NCN2
=====
typecheck((element, *:NCN2) on element QN1 {T}, statEnvs) = (element QN1 {T})
```

A regra a seguir utiliza a comparação das partes locais para fazer a verificação de tipo do filtro para *node test* sem prefixo.

[node-6]

```
NCN1 = NCN2
=====
typecheck((element, *:NCN2) on element *:NCN1 {T}, statEnvs) =
                                                    (element *:NCN2 {T})
```

A regra a seguir se aplica a filtros para *node tests* sem prefixo e nomes de elementos sem parte local.

```
[node-7] typecheck((element, *:NCN2) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:NCN2 {T})
```

As regras a seguir [node-8] a [node-12] fazem a verificação do tipo do filtro `element *:NCN2` sobre um tipo de elemento. Buscas no componente de ambiente estático `namespace` e aplicação da função para obter o prefixo de *QNames* `xs:get-namespace-uri` são aplicadas.

```
[node-8] typecheck((element, *:NCN2) on element {T}, statEnvs) =
                                                    (element *:NCN2 {T})
```

[node-9]

```
xf:get-namespace-uri(QN1) = lookup(NCN2, namespace(statEnvs))
=====
typecheck((element, NCN2:*) on element QN1 {T}, statEnvs) = (element QN1 {T})
```

```
[node-10] typecheck((element, NCN2:*) on element *:NCN1 {T}, statEnvs) =
                                                    (element NCN2:NCN1 {T})
```

[node-11]

```
lookup(NCN1, namespace(statEnvs)) = lookup(NCN2, namespace(statEnvs))
=====
```

```

typecheck((element, NCN2:*) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:* {T})

```

```

[node-12] typecheck((element, NCN2:*) on element {T}, statEnvs) =
                                                    (element NCN2:* {T})

```

Esta última regra relacionada a elementos retorna o mesmo tipo sobre o qual o filtro é aplicado.

```

[node-13] typecheck((element, *) on element Pre:Loc {T}, statEnvs) =
                                                    (element Pre:Loc {T})

```

As regras para atributos são semelhantes às comentadas acima com a diferença de serem aplicadas a atributos. Estas regras são mostradas no Apêndice A.16 nomeadas com rótulos de [node-14] a [node-26].

O tipo de um filtro com *node test* instrução de processamento sobre o tipo instrução de processamento tem como resultado na análise estática o tipo **processing-instruction**, como mostram as regras seguinte.

```

[node-27] typecheck((PNK, processing-instruction()) on processing-instruction,
                                                    statEnvs) = (processing-instruction)

```

```

[node-28] typecheck((PNK, processing-instruction(StringLit))
on processing-instruction, statEnvs) = (processing-instruction)

```

Na regra a seguir, o tipo de um filtro com *node test* comentário sobre o tipo comentário tem como resultado na análise estática o tipo **comment**.

```

[node-29] typecheck((PNK, comment()) on comment, statEnvs) = (comment)

```

O tipo de um filtro com *node test* texto sobre o tipo texto tem como resultado na análise estática o tipo **text**.

```

[node-30] typecheck((PNK, text()) on text, statEnvs) = (text)

```

No caso das regras acima não serem aplicadas, então a sequência vazia é retornada, conforme a última regra.

```

[node-31] typecheck((PNK, node()) on NodeT, statEnvs) = NodeT

```

4.8.3 Expressões de Sequência

A sintaxe para expressões de sequência de XQuery *core* é mostrada na Seção 3.5.3. A verificação de tipos de expressões de sequência envolve a tipagem de cada expressão que compõe a sequência. A seguir, é mostrada a verificação estática de tipos de expressões de sequência, implementada no meta-ambiente ASF+SDF. A verificação de expressões parentetizadas também é mostrada nesta seção.

As regras a seguir estabelecem a verificação de tipos para sequências e expressões entre parênteses.

[sequ]

```
typecheck-list([E, Es], statEnvs) = [Ts]
```

```
=====
```

```
typecheck(E, Es, statEnvs) = [Ts]
```

[paren]

```
typecheck(E, Es, statEnvs) = LT
```

```
=====
```

```
typecheck((E, Es), statEnvs) = LT
```

A regra [paren] especifica que o tipo de uma expressão parentetizada é o tipo da expressão sem os parênteses. A regra [sequ] é a regra que faz a verificação de tipos de sequências, propriamente dita. Para a verificação em *loop* de todas as expressões que compõem a sequência é utilizada uma função `typecheck-list`, implementada nas regras a seguir.

[lst-1] `typecheck-list([], statEnvs) = [()]`

[lst-2]

```
typecheck(E, statEnvs) = T
```

```
=====
```

```
typecheck-list([E], statEnvs) = [T]
```

[lst-3]

```
typecheck(E, statEnvs) = T,
```

```
typecheck-list([Es], statEnvs) = [Ts]
```

```
=====
```

```
typecheck-list([E, Es], statEnvs) = [T, Ts]
```

A função `typecheck-list` está implementada nas regras [1st-1], [1st-2] e [1st-3]. O tipo de uma expressão vazia é o tipo vazio (regra [1st-1]). As regras [1st-2] e [1st-3] estabelecem a busca e verificação de tipos na lista.

4.8.4 Expressões Aritméticas

As regras semânticas estáticas para expressões aritméticas especificam os tipos resultantes para todos os operadores aritméticos, quando aplicados a tipos numéricos específicos. A seguir, é mostrada a verificação estática de tipos de expressões aritméticas, implementada no meta-ambiente ASF+SDF.

A regra original da semântica de XQuery diz que a operação entre duas expressões resulta no tipo decimal quando os tipos das duas expressões são subtipos do tipo decimal.

$$\begin{array}{c}
 \text{statEnvs} \vdash \text{Expr}_1 : \text{Type}_1 \\
 \text{Type}_1 <: \text{xs:decimal} \\
 \text{statEnvs} \vdash \text{Expr}_2 : \text{Type}_2 \\
 \text{Type}_2 <: \text{xs:decimal} \\
 \hline
 \text{statEnvs} \vdash \text{op} : \text{operation}(\text{Expr}_1, \text{Expr}_2) : \text{xs:decimal}
 \end{array}$$

Esta regra, traduzida no meta-ambiente, é mostrada a seguir.

[arith-1]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:decimal, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:decimal, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:decimal

```

A regra a seguir diz que operação entre duas expressões resulta no tipo *float* quando os tipos das duas expressões são subtipos do tipo *float*.

[arith-2]

```

typecheck(E1, statEnvs) = T1,

```

```

subtype(T1, xs:float, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:float, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:float

```

A operação entre duas expressões resulta no tipo *double* quando os tipos das duas expressões são subtipos do tipo *double*.

[arith-3]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:double, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:double, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:double

```

A operação entre duas expressões resulta no tipo inteiro quando os tipos das duas expressões são subtipos do tipo inteiro.

[arith-4]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:integer, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:integer, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:integer

```

No caso de operadores unários as regras são semelhantes. A operação entre uma expressão resulta no tipo decimal quando o tipo da expressão é subtipo do tipo decimal.

[arith-5]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:decimal, statEnvs) = true

```

```
=====
typecheck(op:operation(E1), statEnvs) = xs:decimal
```

A operação entre uma expressão resulta no tipo *float* quando o tipo da expressão é subtipo do tipo *float*.

[arith-6]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:float, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:float
```

A operação entre uma expressão resulta no tipo *double* quando o tipo da expressão é subtipo do tipo *double*.

[arith-7]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:double, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:double
```

A operação entre uma expressão resulta no tipo inteiro quando o tipo da expressão é subtipo do tipo inteiro.

[arith-8]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:integer, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:integer
```

Em [MMRW02] os operadores aritméticos estão definidos. Neste projeto podem ser utilizados os operadores de binários `op:numeric-add` e `op:numeric-subtract` e o operador unário `op:numeric-unary-minus`. Esses operadores substituem `op:operation`.

4.8.5 Construtores

A sintaxe para construtores de XQuery *core* é mostrada na Seção 3.5.5. Somente há regras para verificação de tipos de `ComputedElementConstructor` e `ComputedAttributeConstructor`. A seguir, é mostrada a verificação estática de tipos para tais construtores, implementada no meta-ambiente ASF+SDF.

A análise de tipos estática de construtores utiliza a função `expand` descrita em 4.5.1 e implementada no módulo `StatEnvironment` (Seção 4.5.1).

Na regra a seguir, o tipo para o construtor elemento é obtido através da aplicação da função `expand` ao nome do elemento.

[const-1]

```
expand(QN, statEnvs) = qname
=====
typecheck(element QN {ExprSeq}, statEnvs) = element qname {xs:anyType}
```

Na regra a seguir, o tipo para o construtor elemento com expressão é obtido através da verificação do tipo da expressão E.

[const-2]

```
typecheck(E, statEnvs) = xs:NCN
=====
typecheck(element {E} {ExprSeq}, statEnvs) = element {xs:anyType}
```

Na regra a seguir, o tipo para o construtor atributo com expressão é obtido através da verificação do tipo da expressão E.

[const-3]

```
typecheck(E, statEnvs) = xs:NCN
=====
typecheck(attribute {E} {ExprSeq}, statEnvs) = attribute {xs:anySimpleType}
```

Nas primeira das duas regras a seguir, o tipo para o construtor atributo é obtido através da aplicação da função `expand` ao nome do elemento. Na segunda regra, é estabelecido o padrão de resultado para verificação de tipos de construtores.

[const-4]

expand(QN, statEnvs) = qname

=====

typecheck(attribute QN {ExprSeq}, statEnvs)

= attribute qname {xs:anySimpleType}

[default-const] typecheck(Constr, statEnvs) = static error

4.8.6 Expressões FLWR

A sintaxe para expressões FLWR XQuery *core* é mostrada na Seção 3.5.6. Somente há regras estáticas para expressões *for* e expressões *let*. A seguir, é mostrada a verificação estática de tipos para expressões FLWR, implementada no meta-ambiente ASF+SDF.

A especificação formal segundo a semântica de XQuery para expressões *For* é mostrada a seguir.

$$\begin{array}{c}
 \text{statEnvs} \vdash \text{Expr}_1 : \text{Type} \\
 \hline
 \text{statEnvs}[\text{varType}(\text{Variable}_1 : \text{prime}(\text{Type}_1))] \vdash \text{Expr}_2 : \text{Type}_2 \\
 \hline
 \text{statEnvs} \vdash \text{for } \text{Variable}_1 \text{ in } \text{Expr}_1 \text{ return } \text{Expr}_2 : \text{Type}_2.\text{quantifier}(\text{Type}_1)
 \end{array}$$

$$\begin{array}{c}
 \text{statEnvs} \vdash \text{Expr}_1 : \text{Type} \\
 \text{Type}_0 = [\text{SequenceType}]_{\text{sequencetype}} \\
 \text{Type}_1 <: \text{Type}_0 \\
 \hline
 \text{statEnvs}[\text{varType}(\text{Variable}_1 : \text{prime}(\text{Type}_1))] \vdash \text{Expr}_2 : \text{Type}_2 \\
 \hline
 \text{statEnvs} \vdash \text{for } \text{SequenceType } \text{Variable}_1 \text{ in } \text{Expr}_1 \text{ return } \text{Expr}_2 : \text{Type}_2.\text{quantifier}(\text{Type}_1)
 \end{array}$$

A seguir são mostradas as regras resultado da tradução para o meta-ambiente ASF+SDF.

[for-1]

typecheck(E1, statEnvs) = T1,

insert(\$ Var1 --> prime(T1)) in VT of statEnvs = statEnvs1,

typecheck(TExpr, statEnvs1) = T2

=====

typecheck(for \$ Var1 in E1 return TExpr, statEnvs) = T2.quantifier(T1)

[for-2]

```

typecheck(E1, statEnvs) = T1,
T0 = normSeqT(SeqT),
subtype(T1, T0, statEnvs) = true,
insert($ Var1 --> prime(T1)) in VT of statEnvs = statEnvs1,
typecheck(TExpr, statEnvs1) = T2
=====
typecheck(for SeqT $ Var1 in E1 return TExpr, statEnvs) = T2.quantifier(T1)
[default-for] typecheck(For, statEnvs) = static error

```

A regra [for-1] é aplicada quando não há uma seqüência de tipos SeqT presente. Inicialmente, a primeira expressão E1 é verificada. A função de inclusão de pares no ambiente estático insert é então utilizada para inserir a variável da expressão For no componente de ambiente var-type, mapeada ao *prime type* do tipo da expressão E1. Finalmente, o tipo da expressão de retorno é verificado. Como resultado da expressão for tem-se T2.quantifier(T1).

A regra [for-2] é aplicada quando há uma seqüência de tipos SeqT presente. Basicamente, há duas diferenças entre esta regra e [for-1]. A primeira é a chamada à função de normalização para a variável de seqüência de tipos (normSeqT(SeqT)). A segunda diferença é a verificação de subtipos entre o tipo da primeira expressão E1 e o tipo retornado pela normalização T0. Como resultado da expressão for tem-se T2.quantifier(T1).

A expressão de retorno da expressão For estava definida (como mostram as regras originais de XQuery no início desta seção) como Expr2. Contudo, verificando a sintaxe do comando for constatou-se que não é qualquer expressão que pode figurar neste local; somente são permitidas expressões Typeswitch, ou seja, expressões condicionais, de cast, de caminho, construtores, e assim por diante. Novas ocorrências de expressões FLWR, por exemplo, somente são permitidas se vierem acompanhadas de parênteses (ParenthesizedExpr). Esta correção foi feita.

As regras para let são semelhantes às regras para for.

[let-1]

```

typecheck(E1, statEnvs) = T1,
insert($ Var1 --> T1) in VT of statEnvs = statEnvs1,
typecheck(TExpr, statEnvs1) = T2
=====

```

```

typecheck(let $ Var1 := E1 return TExpr, statEnvs) = T2
[let-2]
typecheck(E1, statEnvs) = T1,
T0 = normSeqT(SeqT),
subtype(T1, T0, statEnvs) = true,
insert($ Var1 --> T1) in VT of statEnvs = statEnvs1,
typecheck(TExpr, statEnvs1) = T2
=====
typecheck(let SeqT $ Var1 := E1 return TExpr, statEnvs) = T2
[default-let] typecheck(Let, statEnvs) = static error

```

A diferença básica é que a variável da cláusula `let` é incluída no ambiente `var-type` associada ao tipo (T1) da primeira expressão e não ao *prime type* dele, como no caso de `for`.

As regras originais conforme a semântica formal de XQuery são mostradas a seguir.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{Expr}_1 : \text{Type}_1 \\
\hline
\text{statEnvs}[\text{varType}(\text{Variable}_1 : \text{Type}_1)] \vdash \text{Expr}_2 : \text{Type}_2 \\
\hline
\text{statEnvs} \vdash \text{let } \text{Variable}_1 := \text{Expr}_1 \text{ return } \text{Expr}_2 : \text{Type}_2
\end{array}$$

$$\begin{array}{c}
\text{statEnvs} \vdash \text{Expr}_1 : \text{Type} \\
\text{Type}_0 = [\text{SequenceType}]_{\text{sequencetype}} \\
\text{Type}_1 <: \text{Type}_0 \\
\hline
\text{statEnvs}[\text{varType}(\text{Variable}_1 : (\text{Type}_1))] \vdash \text{Expr}_2 : \text{Type}_2 \\
\hline
\text{statEnvs} \vdash \text{let } \text{Variable}_1 \text{ in } \text{Expr}_1 \text{ return } \text{Expr}_2 : \text{Type}_2
\end{array}$$

Note que a segunda regra não faz referência a `SequenceType` do lado direito da conclusão. Isto faz com que a regra nunca seja aplicada, pois nas premissas a verificação da normalização de `SequenceType` ocorre, mas esta variável não é fornecida, o que torna uma premissa sempre falsa. Esta modificação foi feita no protótipo.

4.8.7 Expressões de Ordem

A sintaxe para expressões de ordem XQuery *core* é mostrada na Seção 3.5.7. A seguir, é mostrada a verificação estática de tipos para expressões de ordem, implementada no meta-

ambiente ASF+SDF.

[unord]

```
typecheck(FLEExpr, statEnvs) = T1
```

```
=====
```

```
typecheck((unordered FLEExpr), statEnvs) = prime(T1).quantifier(T1)
```

[default-unord] typecheck(UnorExpr, statEnvs) = static error

A regra estática para expressões *unordered* especifica a tipagem através da verificação de tipos para a expressão FLEExpr da expressão *unordered*. A regra original é mostrada a seguir.

$$\frac{\text{statEnvs} \vdash \text{Expr}_1 : \text{Type}_1}{\text{statEnvs} \vdash \text{unordered Expr} : \text{prime}(\text{Type}_1).\text{quantifier}(\text{Type}_1)}$$

Analisando as regras gramaticais para UnorderedExpr verifica-se que não é qualquer expressão XQuery *core* que pode ser utilizada onde figura *Expr*. Então *Expr* foi substituído por *FLEExpr*, que é uma variável definida para aceitar ou expressões *for* ou expressões *let* e expressões derivadas destas na árvore gramatical de XQuery *core*. Esta modificação foi feita no protótipo.

4.8.8 Expressões Condicionais

A sintaxe para expressões condicionais XQuery *core* é mostrada na Seção 3.5.8. A seguir, é mostrada a verificação estática de tipos para expressões condicionais, implementada no meta-ambiente ASF+SDF.

[if]

```
typecheck(E1, statEnvs) = xs:boolean,
```

```
typecheck(E2, statEnvs) = T2,
```

```
typecheck(ValExpr, statEnvs) = T3
```

```
=====
```

```
typecheck((if (E1) then E2 else ValExpr), statEnvs) = (T2 | T3)
```

A regra de análise estática [if] especifica como tipagem da expressão condicional uma alternativa entre dois tipos (T2 e T3). A principal condição para que isso aconteça é que primeira

expressão do `if` deve ser booleana. A verificação de tipos determina para a expressão condicional ou o tipo da segunda expressão (`E2`) ou o tipo da terceira expressão (`ValExpr`). A terceira expressão (após o `else`) é uma variável de `ValueExpr`, isto é, pode assumir expressões de validação, de `cast`, de caminho, construtores ou expressões derivadas destas na árvore gramatical de XQuery *core*. Originalmente estava, nesta posição, uma variável `Expr`, como pode ser verificado a seguir.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{Expr}_1 : \text{xs} : \text{boolean} \\
\text{statEnvs} \vdash \text{Expr}_2 : \text{Type}_2 \\
\text{statEnvs} \vdash \text{Expr}_3 : \text{Type}_3 \\
\hline
\text{statEnvs} \vdash \text{if } (\text{Expr}_1) \text{ then } \text{Expr}_2 \text{ else } \text{Expr}_3 : (\text{Type}_2 \mid \text{Type}_3)
\end{array}$$

Pela própria gramática da linguagem XQuery *core* não é qualquer expressão que pode ser usada neste local. Para respeitar a construção da gramática de XQuery *core*, esta modificação foi efetuada no protótipo.

4.8.9 Expressões sobre Sequência de Tipos (*cast*)

A sintaxe para expressões sobre sequência de tipos XQuery *core* é mostrada na Seção 3.5.9. A seguir, é mostrada a verificação estática de tipos para expressões sobre sequência de tipos para *cast*, implementada no meta-ambiente ASF+SDF.

Para a análise de expressões de *cast* é necessária a consulta a uma tabela com os tipos onde o *cast* é permitido (Y), não é permitido (N) ou pode ser feito em alguns casos (M). Se a tabela de *cast* indica que o *cast* não é permitido, o sistema aponta um erro de tipo estático. Caso contrário, as regras abaixo retornam a semântica estática destas expressões.

As regras que determinam a análise estática para expressões *cast* utilizam uma notação auxiliar:

$$\text{Type}_1 \text{ cast allowed } \text{Type}_2 \Rightarrow Y, M, N$$

Esta notação especifica se o *cast* de um tipo para outro é permitido. A tabela que determina esta conversão é descrita na Seção 4.6.2 e está implementada no módulo `CastingTable`.

Se o tipo estático da expressão Type_1 pode ser a sequência vazia, então o tipo estático do *cast* é o opcional de Type_2 . Esta regra conforme a semântica formal de XQuery é mostrada a

seguir.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{Expr} : \text{Type}_1 \\
\text{quantifier}(\text{Type}_1) = ? \text{ or } \text{quantifier}(\text{Type}_1) = * \\
\text{Type}_2 = [\text{SequenceType}_2]_{\text{sequencetype}} \\
\hline
\text{Type}_1 \text{ cast allowed } \text{Type}_2 \Rightarrow Y \text{ or } \text{Type}_1 \text{ cast allowed } \text{Type}_2 \Rightarrow M \\
\text{statEnvs} \vdash \text{cast as } \text{SequenceType}_2 (\text{Expr}) : \text{Type}_2?
\end{array}$$

Para ser utilizada pelo meta-ambiente ASF+SDF, esta regra foi traduzida das quatro regras a seguir.

[cast-1a]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = ?,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1b]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1c]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = ?,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1d]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = *,

```

```

T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

As regras [cast-1a] a [cast-1d] fazem a verificação se T1 pode ser uma sequência vazia (através dos símbolos ? ou *). Nesse caso, o tipo estático de saída é T2 opcional.

Se o tipo estático da expressão $Type_1$ não pode ser a sequência vazia, então o tipo estático do *cast* é $Type_2$. Esta regra conforme a semântica formal de XQuery é mostrada a seguir.

$$\begin{array}{c}
\text{statEnvs} \vdash \text{Expr} : Type_1 \\
\text{quantifier}(Type_1) \neq ? \text{ quantifier}(Type_1) \neq * \\
Type_2 = [SequenceType_2]_{sequence\ type} \\
\hline
Type_1 \text{ cast allowed } Type_2 \Rightarrow Y \text{ or } Type_1 \text{ cast allowed } Type_2 \Rightarrow M \\
\text{statEnvs} \vdash \text{cast as } SequenceType_2 (\text{Expr}) : Type_2
\end{array}$$

Para ser utilizada pelo meta-ambiente ASF+SDF, esta regra foi traduzida das quatro regras a seguir.

[cast-2a]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) != ?,
quantifier(T1) != *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = T2

```

[cast-2b]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) != ?,
quantifier(T1) != *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = T2

```

A regra [cast-2a] e a regra [cast-2b] fazem a verificação se T1 não é uma sequência vazia (através dos símbolos ? ou *). Nesse caso, o tipo estático de saída é T2.

Como visto, no documento de semântica formal, CastExpr possui somente duas regras para análise estática de tipos. A primeira regra do documento resultou nas regras [cast-1a], [cast-1b], [cast-1c] e [cast-1d] do meta-ambiente ASF+SDF. A segunda regra do documento resultou nas regras [cast-2a] e [cast-2b]. Isto aconteceu devido a exigências do meta-ambiente com relação à comparação dos padrões dos *sorts*.

4.8.10 Observações Finais

Algumas expressões e construções XQuery não têm seus correspondentes em XQuery *core* e, portanto, não tem semântica formal. Outras, têm gramática *core*, mas não possuem regras de análise de tipo estática. Dentre estes, destacam-se:

- expressões de comentário;
- predicados;
- sintaxe abreviada e não abreviada;
- expressões de comparação;
- expressões lógicas;
- expressões quantificadas;
- expressões de validação.

Expressões de validação, embora não possuam regras de validação estática, podem ser avaliadas dinamicamente.

4.9 Funcionamento do Ambiente do Protótipo

A seguir, são apresentados os passos para usar o interpretador gerado para o protótipo:

1. Ir até o diretório onde estão todos os módulos do protótipo.
2. Digitar o comando `meta`, que chama o meta-ambiente ASF+SDF. A janela principal aparecerá como mostra a Figura 4.1.
3. Selecionar o menu *File/Open* e escolher o módulo `XQueryStaticTyping` na janela de diálogo. Após confirmar a operação, o ASF+SDF carregará este módulo (que é o módulo principal da aplicação) para o sistema.
4. Verificar que todos os módulos do protótipo devem aparecer, conforme mostra a Figura 4.1.
5. Selecionar o módulo `XQueryStaticTyping` no meta-ambiente, clicar sobre ele com o botão direito e escolher *Edit Term*. Entrar com um nome de arquivo, por exemplo, *Expressoes.trm*.
6. Digitar o termo (expressão) a ser testado.
7. Selecionar o menu *Actions/Parse* para verificar se o termo está sintaticamente correto.
8. Selecionar o menu *Actions/Reduce* para reduzir o termo. Este resultado aparecerá em outra janela do editor.

O termo reduzido é o resultado da verificação do tipo da expressão fornecida.

4.10 Usando o Protótipo

Esta seção aborda algumas situações de testes a que o protótipo foi submetido. A entrada é constituída de expressões XQuery *core* e valores para o ambiente estático. A saída é o tipo (ou tipos) das expressões. Por ainda ser uma especificação em desenvolvimento e, portanto, incompleta, algumas expressões de XQuery *core* ainda não podem ser testadas ou podem apenas ser testadas parcialmente.

No primeiro exemplo, serão mostradas algumas das regras utilizadas ao longo da verificação de tipos de expressões XQuery *core*, simulando alguns dos passos seguidos pelo meta-ambiente ASF+SDF.

Segue um exemplo da verificação de tipos utilizando expressões *For* e *If* de XQuery *core* em conjunto com uma chamada de função:

```
typecheck(
  for $b in
    (if (eq($v1,$v2)) then $v1 else $root)
  return $dot,
  [],[],[],[],[],
  [($dot --> xs:decimal) ($b --> xs:integer)
  ($root --> xs:decimal) ($r --> xs:string)
  ($v1 --> xs:decimal) ($v2 --> xs:decimal)],
  [((eq) --> define function eq(xs:decimal, xs:decimal) returns xs:boolean)],
  [])
```

A regra básica (para a expressão principal *for*) utilizada para a verificação de tipos é a seguinte:

```
[for-1] typecheck(E1, statEnvs) = T1,
        insert($ Var1 --> prime(T1)) in VT of statEnvs = statEnvs1,
        typecheck(TExpr, statEnvs1) = T2
        =====
        typecheck(for $ Var1 in E1 return TExpr, statEnvs) =
                                T2.quantifier(T1)
```

A regra para verificação da expressão *for* diz que a primeira expressão (E1) deve ser verificada, uma inserção no ambiente deve ser feita e a expressão de retorno (TExpr) deve ser verificada. Inicialmente, a verificação da primeira expressão (no caso do exemplo, a expressão condicional) deve ser feita. A regra utilizada é mostrada a seguir:

```
[if] typecheck(E1, statEnvs) = xs:boolean,
      typecheck(E2, statEnvs) = T2,
      typecheck(ValExpr, statEnvs) = T3
      =====
      typecheck((if (E1) then E2 else ValExpr), statEnvs) = (T2 | T3)
```

A regra para verificação da expressão *if* determina que outras chamadas à função *typecheck*

sejam feitas. Três verificações de tipos são necessárias. A primeira delas utiliza a chamada de função (eq), testando se retorna um tipo booleano. A regra para tal é mostrada a seguir:

```
[func] FuncCall = QN(E1, Es),
    lookup((QN), func-type(statEnvs)) =
        define function QN(T1, Ts) returns T,
    typecheck-list([E1, Es], statEnvs) = [T2, Ts2],
    subtype-list([T2, Ts2], [T1, Ts], statEnvs) = true
    =====
    typecheck(FuncCall, statEnvs) = T
```

Esta regra faz uma busca no ambiente utilizando a função lookup (Seção 4.5.1), faz uma verificação de tipos dos parâmetros da chamada da função (Seção 4.8.3) e busca relacionamentos de subtipos (Seção 4.4.2).

Após estas verificações, volta-se à regra para a expressão condicional e verificam-se os tipos de variáveis. A regra para variáveis utilizada é a seguinte:

```
[var]
    lookup($ Var, var-type(statEnvs)) = T
    =====
    typecheck($ Var, statEnvs) = T
```

Esta regra busca no ambiente var-type o valor de tipo para variáveis. No caso do exemplo, \$b, \$v1, \$v2 e \$root são verificadas até este ponto.

Finalmente, para a conclusão da verificação da expressão for, a função insert é utilizada (Seção 4.5.1) e a verificação do tipo para a variável \$dot é feita. O tipo resultante da expressão for, retornado pela função de verificação de tipos, é:

```
[xs:decimal.quantifier(xs:decimal)]
```

Alguns exemplos são brevemente comentados a seguir e sua execução pelo meta-ambiente ASF+SDF é semelhante à descrita acima.

No exemplo a seguir, a expressão *For* de XQuery core é utilizada.

```

typecheck(
  for $b in
    for $dot in $root
      return $v1
  return $dot,
  [], [], [], [], [],
  [($b --> ()) ($dot --> none) ($root --> xs:float*)
  ($r --> xs:string*) ($v1 --> xs:double*) ($v2 --> xs:QName*)],
  [], [])

```

O primeiro parâmetro de `typecheck` é formado por duas expressões *For* aninhadas. O parâmetro seguinte é o ambiente `statEnvs` visualizado através dos seus oito pares. Neste exemplo, os cinco primeiros componentes do ambiente e os dois últimos estão vazios. Estes componentes são: `namespace`, `default-namespace`, `type-defn`, `elem-decl`, `attr-decl`, `func-type` e `base-uri`. O único componente preenchido é o `var-type`, que possui algumas variáveis mapeadas a tipos. A variável `$dot`, por exemplo, está mapeada ao tipo `none`. O tipo para estas expressões, retornado pela função de verificação de tipos, é:

```
[none*]
```

O mesmo exemplo, com algumas modificações nos valores de tipos das variáveis do ambiente é mostrado a seguir.

```

typecheck(
  for $b in
    for $dot in $root
      return $v1
  return $dot,
  [], [], [], [], [],
  [($b --> xd:decimal) ($dot --> xs:string) ($root --> xs:float*)
  ($r --> xs:string*) ($v1 --> xs:double*) ($v2 --> xs:QName*)],
  [], [])

```

Neste caso, o tipo para as expressões, retornado pela função de verificação de tipos, é:

[xs:string*]

No exemplo a seguir, as expressões *For* e *If* de XQuery *core* são utilizadas.

```
typecheck(  
  for $b in  
    (if ($v2) then $dot else $r)  
  return $dot,  
  [],[],[],[],[],  
  [($b --> xd:decimal?) ($dot --> xs:string) ($root --> xs:float?)  
    ($r --> xs:string*) ($v1 --> xs:double?) ($v2 --> xs:boolean)],  
  [],[])
```

O primeiro parâmetro de `typecheck`, neste caso, é formado por duas expressões de XQuery *core*: *For* e *If*. O tipo para estas expressões, retornado pela função de verificação de tipos, é:

[xs:string*]

O exemplo a seguir é semelhante aos anteriores, com algumas alterações nos tipos das variáveis.

```
typecheck(  
  for $b in  
    (if (eq($v1,$v2)) then $v1 else $root)  
  return $dot,  
  [],[],[],[],[],  
  [($dot --> xs:decimal) ($b --> xs:integer)  
    ($root --> xs:string) ($r --> xs:string)  
    ($v1 --> xs:decimal) ($v2 --> xs:decimal)],  
  [((eq) --> define function eq(xs:decimal, xs:decimal) returns xs:boolean)],  
  [])
```

Quando as duas expressões correspondentes às opções `then` e `else` da expressão condicional possuem valores de tipos diferentes, o tipo de retorno da função `typecheck`, é:

[xs:decimal.quantifier(xs:decimal) | quantifier(xs:string)]

Os resultados de tipos retornados pelas verificações utilizando a função `typecheck` estão dentro do esperado, de acordo com as regras especificadas. Regras estas, implementadas em conformidade com a semântica formal de XQuery *core*, sugerida pelo documento que a especifica [DFFM02].

Capítulo 5

Conclusões

Este trabalho apresentou a implementação de um protótipo para a verificação estática de tipos da linguagem XQuery, de consulta a documentos XML, usando a ferramenta ASF+SDF. O protótipo aqui descrito foi construído com base na semântica operacional da linguagem.

Por ser um trabalho em desenvolvimento, a semântica da linguagem de consulta XQuery possui diversos pontos (mais de 170) que precisam ser trabalhados, conforme o documento de semântica formal da linguagem [DFFM02], como por exemplo:

- Especificação incompleta de conversões de tipo.
- Semântica para construtores de elementos e atributos.
- Semântica de *sortby*.
- Tratamento de erros em geral.
- Análise estática de tipos para expressões de validação.
- Regras de equivalência de tipos.
- Tipificação para cláusulas *typeswitch default*.

Durante o processo de implementação do protótipo alguns erros e inconsistências foram encontrados no documento citado e precisaram ser tratados. Estes pontos problemáticos têm

origens diversas sendo desde sintaxe inconsistente com as expressões utilizadas até regras de semântica incompletas. Dentre esses pontos podemos citar:

- No sistema de tipos de XQuery, a inclusão do valor *error* como regra de produção para os não-terminais *Value* e *Type* (Seção 3.4). Esta inclusão se justifica pelo fato de que ao longo da especificação das regras que compõem a análise de tipos de XQuery *core* (de acordo com [DFFM02]) o valor de erro é utilizado sem, contudo, fazer parte da gramática da linguagem. O tratamento de erros para XQuery ainda não está definido pelo W3C.
- No sistema de tipos de XQuery, a inclusão do não-terminal *Wildcard* como regra de produção para os não-terminais *ElementName* e *AttributeName* (Seção 3.4). Este não-terminal foi adicionado devido a inconsistências nas regras de análise de tipo estática para *node tests* (Seção 4.8.2). Tais regras aplicavam *wildcards* para *ElementName* e *AttributeName*, sem isto ser permitido, contudo, na gramática destes não-terminais.
- A unificação de indicadores de ocorrências (*OccurrenceIndicator*). O sistema de tipos de XQuery e a gramática normalizada de XQuery tinham não-terminais definidos distintamente para indicadores de ocorrências. Estas definições eram semelhantes e, por isso, foram alteradas e unificadas em um só não-terminal, que passou a ser utilizado em toda a aplicação.
- A incorporação dos não-terminais *AttributeAll* e *ElementContent* ao sistema de tipos de XQuery (Seção 3.4). Estes dois não-terminais, embora possuindo produções, não faziam parte do sistema de tipos e nem da gramática de XQuery devido ao fato de que não eram chamados como produções de nenhum outro não-terminal. Eles são usados nas regras que definem o funcionamento das instruções auxiliares (Seção 3.4.5). Nestas regras são usados como se fossem parte da definição de tipos (*ItemType*). Este uso esclareceu sua origem e justificou sua inclusão como produção do não-terminal *ItemType*.
- Inclusão do não-terminal *Type* como produção de *PrimeType* (Seção 3.4.4). As regras semânticas que definem a especificação de *prime types* são mostradas utilizando as produções do não-terminal *Type* sem, contudo, este estar associado a *prime types*. Devido a este fato, o não-terminal *Type* foi incluído como produção de *PrimeType*.

- Inclusão do *sort LetExpr* à gramática de XQuery *core* (Seção 4.3.2). O *sort LetExpr* foi acrescentado como produção de *UnorderedExpr*. Analisando a gramática de XQuery *core* percebeu-se que apareciam produções para o não-terminal *LetExpr* sem, contudo, este ser apontado em nenhum outro lugar como produção de outro não-terminal. Isto fazia com que ele não fizesse parte da linguagem XQuery *core*, mas aparecia, contudo, como parte da linguagem XQuery.
- Outra correção feita foi para o *sort ItemType* da gramática de XQuery *core* (Seção 4.3.2). O sistema de tipos de XQuery (Seção 3.4) também possuía um *sort* definido como *ItemType*. Ao longo da implementação da especificação percebeu-se que o uso deste não-terminal é feito sem mencionar se a referência é sobre o *ItemType* da gramática ou o do sistema de tipos. Assim, assumiu-se que *ItemType* da gramática é o mesmo que *ItemType* do sistema de tipos, unindo suas produções. Para manter a modularidade, está declarado no módulo *XQueryCoreSyntax* e suas produções estão em *XQueryCoreSyntax* e também no módulo *TypeSystem*. Para não gerar ambigüidade, foram retirados *text* e *untyped* da produção de *ItemType*, porque estes símbolos já estavam definidos em *TypeSystem*.
- Inclusão do ambiente local de tipos para produções de *axis* descendentes. As regras [axes-6] e [axes-7] do módulo *XQueryStaticTyping* foram modificadas das suas formas originais no documento de semântica formal de XQuery. Estas regras, juntamente com a [axes-8], computam o tipo estático de *axis* descendentes utilizando o ambiente adicional *localTypeEnv*. Contudo, embora as regras [axes-6] e [axes-7] estivessem em suas premissas utilizando tal ambiente, este não era passado a elas. A correção feita foi a adição de *localTypeEnv* ao lado esquerdo da conclusão a fim de torná-lo utilizável pelas regras.
- Especificação de regras para *axes*. A aplicação de filtro para *axes* (Seção 4.8.2) possuía uma regra geral que resulta em um tipo vazio, caso as regras específicas definidas não fossem utilizadas. A regra geral era ambígua para o meta-ambiente ASF+SDF então, os casos restantes foram especificados um a um no protótipo, conforme a seguir.

[axes-11] `typecheck(following-sibling:: on NodeT, statEnvs) = ()`

[axes-12] `typecheck(following:: on NodeT, statEnvs) = ()`

[axes-13] `typecheck(preceding-sibling:: on NodeT, statEnvs) = ()`

[axes-14] `typecheck(preceding:: on NodeT, statEnvs) = ()`

- Modificação da regra [node-1] (Seção 4.8.2) original da semântica de XQuery. A regra estabelecia uma comparação de igualdade entre o resultado da função `xf:get-namespace-uri` e o resultado de uma busca no ambiente de `statEnvs var-type (lookup)`. Esta igualdade nunca seria verdadeira, porque o ambiente `var-type` mapeia nomes de variáveis a seus tipos, o que não pode ser igual a uma URI, que é o que a função `xf:get-namespace-uri` retorna. O ambiente correto a ser utilizado neste caso é o `namespace` de `statEnvs`. Tal alteração foi efetuada no protótipo.
- A expressão de retorno da expressão *For* estava definida como `Expr2` (Seção 4.8.6). Contudo, verificando a sintaxe do comando `for` constatou-se que não é qualquer expressão que pode figurar neste local; somente são permitidas expressões `Typeswitch`, ou seja, expressões condicionais, de *cast*, de caminho, construtores, e assim por diante. Novas ocorrências de expressões FLWR, por exemplo, somente são permitidas se vierem acompanhadas de parênteses (`ParenthesizedExpr`). Esta correção foi feita.
- A segunda regra para expressões *Let* não fazia referência a `SequenceType` do lado direito da conclusão (Seção 4.8.6). Isto fazia com que a regra nunca fosse aplicada, pois nas premissas a verificação da normalização de `SequenceType` ocorria, mas esta variável não era fornecida, o que tornava uma premissa sempre falsa. Esta modificação foi feita no protótipo.
- Analisando as regras gramaticais para `UnorderedExpr` (Seção 4.8.6) verificou-se que não é qualquer expressão XQuery *core* que pode ser utilizada onde figura *Expr* na regra. Então *Expr* foi substituído por *FLEpr*, que é uma variável definida para aceitar ou expressões *for* ou expressões *let* e expressões derivadas destas na árvore gramatical de XQuery *core*. Esta modificação foi feita no protótipo.
- Na regra para análise estática de expressões condicionais (Seção 4.8.8), a terceira expressão (após o `else`) é uma variável de `ValueExpr`, isto é, pode assumir expressões de validação, de *cast*, de caminho, construtores ou expressões derivadas destas na árvore gramatical

de XQuery *core*. Originalmente estava, nesta posição, uma variável *Expr*. Pela própria gramática da linguagem XQuery *core* não é qualquer expressão que pode ser usada neste local. Para respeitar a construção da gramática de XQuery *core*, esta modificação foi efetuada no protótipo.

A descoberta destes pontos, assim como a implementação da verificação estática de tipos, formam a contribuição deste trabalho.

Os resultados de tipos retornados pelas verificações utilizando o protótipo estão dentro do esperado, de acordo com as regras semânticas implementadas em conformidade com a semântica formal de XQuery *core*.

A adaptação das regras originais da semântica operacional da linguagem para a forma em que o sistema ASF+SDF exige foi trabalhosa. Isto se justifica pelo fato de que a semântica operacional da linguagem não foi elaborada visando a implementação imediata da mesma.

A semântica operacional é uma das técnicas para descrição formal que permite, entre outras coisas, a formalização de linguagens. Esta formalização contribuiu neste trabalho para perceber os detalhes da linguagem de consulta XQuery, entender sua forma de funcionamento e, por ainda ser uma especificação em desenvolvimento, perceber as inconsistências de sua semântica formal.

5.1 Trabalhos Futuros

Existem vários pontos em aberto relacionados à semântica operacional de XQuery. Alguns dentre os de maior relevância são citados a seguir:

- atualização do protótipo utilizando novas versões do documento de semântica formal de XQuery;
- implementação de expressões sobre sequência de tipos *typeswitch*;
- implementação da importação de *schemas* e normalização;
- implementação do contexto de processamento de *Query Prolog*;
- implementação da avaliação dinâmica.

Estes itens citados envolvem o estudo e implementação de especificações relacionadas ao documento de semântica formal de XQuery e a outros documentos publicados pelo W3C. O desenvolvimento destes trabalhos apresenta dificuldade especialmente por utilizar documentos e linguagens em construção.

Bibliografia

- [Abi97] Serge Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997. citeseer.nj.nec.com/abiteboul97querying.html.
- [ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Gerenciando Dados na Web*. Campus, Rio de Janeiro, 2000.
- [Acz77] Peter Aczel. *An Introduction to Inductive Definitions*. North-Holland Publishing Company, North-Holland, 1977.
- [AQM⁺97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997. citeseer.nj.nec.com/article/abiteboul97lorel.html.
- [Ast91] Edigio Astesiano. Inductive and operacional semantics. In *Formal Description of Programming Concepts*, 1991.
- [ASU88] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1988.
- [BBBD00] Mark Baartse, Richard Blair, Luca Bolognese, and Dinar Dalvi. *Professional ASP XML*. Wrox Press, 2000.
- [BC00] Angela Bonifati and Stefano Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000. citeseer.nj.nec.com/bonifati00comparative.html.

- [BCF⁺02] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Simeon, and Mugur Stefanescu. Xquery 1.0: An xml query language. Working draft, W3C World Wide Web Consortium, april 2002. www.w3.org/TR/2002/WD-xquery-20020430.
- [BHL99] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in xml. Technical report, W3C World Wide Web Consortium, january 1999. www.w3.org/TR/1999/REC-xml-names-19990114.
- [BK03] M. G. J. Van Den Brand and P. Klint. Asf+sdf meta-environment user manual - revision 1.134. Technical report, CWI Centrum voor Wiskunde en Informatica, Amsterdam, 2003. www.cwi.nl/projects/MetaEnv/meta.
- [BM01] Paul V. Biron and Ashok Malhotra. Xml schema part 2: Datatypes. Technical report, W3C World Wide Web Consortium, may 2001. www.w3.org/TR/2001/REC-xmlschema-2-20010502.
- [BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (xml) 1.0. Technical report, W3C World Wide Web Consortium, october 2000. www.w3.org/TR/2000/REC-xml-20001006.
- [Bun97] Peter Buneman. Semistructured data. In *16th ACM Symposium on Principles of Database Systems*, pages 117–121, 1997. citeseer.nj.nec.com/buneman97semistructured.html.
- [CCD⁺99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A graphical language for querying and restructuring XML documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165, 1999. citeseer.nj.nec.com/ceri99xmlgl.html.
- [CRF01] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, 1997, 2001. citeseer.nj.nec.com/chamberlin00quilt.html.

- [DeR98] Steven J. DeRose. Xquery: A unified syntax for linking and querying general xml documents. In *Query Languages Workshop (QL98)*, 1998. www.w3.org/TandS/QL/QL98/pp/xquery.html.
- [DFF⁺99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, 1999. citeseer.nj.nec.com/deutsch98query.html.
- [DFFM02] Denise Draper, Peter Fankhauser, Mary Fernadez, and Ashok Malhotra. Xquery 1.0 and xpath 2.0 formal semantics. Working draft, W3C World Wide Web Consortium, august 2002. www.w3.org/TR/2002/WD-query-semantics-20020816.
- [Fal01] David C. Fallside. Xml schema part 0: Primer. Technical report, W3C World Wide Web Consortium, may 2001. www.w3.org/TR/2001/REC-xmlschema-0-20010502.
- [FMN02] Mary Fernandez, Jonathan Marsh, and Marton Nagy. Xquery 1.0 and xpath 2.0 data model. Working draft, W3C World Wide Web Consortium, april 2002. www.w3.org/TR/2002/WD-query-datamodel-20020430.
- [HHKR92] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF - reference manual*, december 1992. ftp.cwi.nl/pub/gipe/reports/SDFmanual.ps.Z.
- [Mar00] Benoit Marchal. *XML Conceitos e Aplicações*. Berkeley, São Paulo, 2000.
- [MMRW02] Ashok Malhotra, Jim Melton, Jonathan Robie, and Norman Walsh. Xquery 1.0 and xpath 2.0 functions and operators. Technical report, W3C World Wide Web Consortium, august 2002. www.w3.org/TR/2002/WD-xquery-operators-20020816.
- [Mus96] Martin A. Musicante. On the relational semantics of interleaving constructors - phd. thesis. Universidade Federal de Pernambuco, 1996.
- [RLS98] Jonathan Robie, Joe Lapp, and David Schach. Xml query language(xql). In *Query Languages Workshop (QL98)*, 1998. www.w3.org/TandS/QL/QL98/pp/xql.html.

- [Sch86] David A. Schmidt. Denotational semantics: A methodology for language development. Allyn Bacon, 1986.
- [SLR98] David Schach, Joe Lapp, and Jonathan Robie. Querying and transforming xml. In *Query Languages Workshop (QL98)*, 1998. www.w3.org/TandS/QL/QL98/pp/query-transform.html.
- [vD] Arie van Deursen. Introducing asf+sdf using the λ -calculus as example. cite-seer.nj.nec.com/vandeursen94introducing.html.
- [Wat91] David Watt. Programming language syntax and semantics. Prentice Hall International (UK), 1991.

Apêndice A

Arquivos de Código do Protótipo

A.1 Layout.sdf

```
module Layout
exports
lexical syntax
  [\ \t\n]          -> LAYOUT
  "%%" ~[\n]* [\n]  -> LAYOUT
  "%" ~[%\n]+ "%"   -> LAYOUT

context-free restrictions
  LAYOUT? -/- [\ \t\n%]
```

A.2 TerminalsSyntax.sdf

```
module TerminalsSyntax
imports Layout
exports
  sorts BaseChar CombiningChar Extender Digit Letter NCName NCNameChar
         QName Prefix LocalPart StringLiteral IntegerLiteral DecimalLiteral
         DoubleLiteral S Digits PITarget VarName HexDigits CharRef Others
```

Char WhitespaceChar OccurrenceIndicator Str

lexical syntax

```

"\\" ~ [\\n]* "\\"      -> Str
"\"\"\"\"               -> Str
[a-zA-Z]                 -> BaseChar
[\\À\\Á\\Â\\Ã\\È\\É\\Ê\\Ë\\Ì\\Í\\Î\\Ï\\Ñ\\Ò\\Ó\\Ô\\Õ\\Ö\\Ù\\Ú\\Û\\] -> BaseChar
[\\à\\á\\â\\ã\\ä\\è\\é\\ê\\ë\\ì\\í\\î\\ï\\ñ\\ò\\ó\\ô\\õ\\ö\\ù\\ú\\û\\ü\\] -> BaseChar
[\\'\\'\\'~\\'"]          -> CombiningChar
[\\.]                     -> Extender
[0-9]                     -> Digit
[0-9]+                    -> Digits
([0-9] | [a-f] | [A-F])+  -> HexDigits
[\\!\\@\\#\\$\\%\\&\\*\\(\\)\\_\\-\\+\\=\\|\\\\\\\\[\\\\]\\\\{\\\\}\\:\\\\;\\\\<\\\\>\\\\,\\\\.\\\\?\\\\/] -> Others
[\\t] | [\\n] | [\\ ] | Others -> Char
[\\t] | [\\n] | [\\ ]        -> WhitespaceChar
("*" | "+" | "?")         -> OccurrenceIndicator
BaseChar                   -> Letter
"&#" (Digits | ("x" HexDigits)) ";" -> CharRef
Digits                     -> IntegerLiteral
("." Digits) | (Digits "." [0-9]*) -> DecimalLiteral
(("." Digits) | (Digits ("." [0-9]*)?)) ([e] | [E])
                                ("+" | "-")? Digits -> DoubleLiteral
(Letter | "_") (NCNameChar)* -> NCName
Letter | Digit | "." | "-" | "_" -> NCNameChar
CombiningChar | Extender   -> NCNameChar

```

context-free syntax

Str	-> StringLiteral
(Prefix ":")? LocalPart	-> QName
NCName	-> Prefix
NCName	-> LocalPart
WhitespaceChar+	-> S

NCName	-> PITarget
QName	-> VarName

variables

"Pre"[0-9]*	-> Prefix
"Loc"[0-9]*	-> LocalPart
"Var"[0-9]*	-> VarName
"Vars"[0-9]*	-> {"\$" VarName) ", "}*
"QN"[0-9]*	-> QName
"NCN"[0-9]*	-> NCName
"IntegerLit"	-> IntegerLiteral
"DecimalLit"	-> DecimalLiteral
"DoubleLit"	-> DoubleLiteral
"StringLit"	-> StringLiteral
"OI"[0-9]*	-> OccurrenceIndicator

A.3 XQueryCoreSyntax.sdf

module XQueryCoreSyntax

imports TerminalsSyntax TypeInteger

exports

sorts Query QueryProlog QueryBody XMLSpaceDecl DefaultCollationDecl
 NamespaceDecl DefaultNamespaceDecl SchemaImport SubNamespaceDecl
 FunctionDefn Param ParamList ExprSequence Expr SortExpr SortSpecList
 SortModifier UnorderedExpr ForExpr ForClause TypeDeclaration LetExpr
 LetClause TypeswitchExpr CaseClause CastExpr SequenceType AtomicType
 ElemOrAttrType SchemaType IfExpr ValueExpr SchemaContext
 SchemaGlobalContext SchemaContextStep ValidateExpr PathExpr
 RelativePathExpr StepExpr ForwardStep ReverseStep Constructor
 EnclosedExpr XmlComment XmlProcessingInstruction
 ComputedDocumentConstructor ComputedElementConstructor
 ComputedAttributeConstructor PrimaryExpr NumericLiteral Literal

ParenthesizedExpr FunctionCall ForwardAxis ReverseAxis NodeTest
 NameTest Wildcard KindTest ProcessingInstructionTest CommentTest
 TextTest AnyKindTest ItemType

lexical syntax

`((["] ((["] | "~["\"])* ["]) | (["'"] ((["'"] | "~'")* '')) -> StringLiteral`

context-free syntax

`QueryProlog QueryBody -> Query`
`(NamespaceDecl | XMLSpaceDecl | DefaultNamespaceDecl`
`| DefaultCollationDecl | SchemaImport)* FunctionDefn* -> QueryProlog`
`ExprSequence? -> QueryBody`
`"declare" "xmlspace" "=" ("preserve" | "strip") -> XMLSpaceDecl`
`"default" "collation" "=" StringLiteral -> DefaultCollationDecl`
`"declare" "namespace" NCName "=" StringLiteral -> NamespaceDecl`
`((("default" "element") | ("default" "function"))`
`"namespace" "=" StringLiteral -> DefaultNamespaceDecl`
`("import" "schema") (StringLiteral | SubNamespaceDecl |`
`DefaultNamespaceDecl) ("at" StringLiteral)? -> SchemaImport`
`"namespace" NCName "=" StringLiteral -> SubNamespaceDecl`
`("define" "function") QName "(" ParamList? ")" |`
`( ")" "returns" SequenceType)) EnclosedExpr -> FunctionDefn`
`SequenceType "$" VarName -> Param`
`Param ("," Param)* -> ParamList`
`{Expr ","}+ -> ExprSequence{prefer}`
`SortExpr -> Expr{avoid}`
`UnorderedExpr ("stable"? "sort" "by" "("`
`SortSpecList ")" )? -> SortExpr`
`Expr SortModifier ("," SortSpecList)? -> SortSpecList`
`("ascending" | "descending")`
`((("empty" "greatest") | ("empty" "least"))) -> SortModifier`
`"unordered"? (ForExpr | LetExpr)+ -> UnorderedExpr`

(ForClause "return")* TypeswitchExpr	-> ForExpr{prefer}
"for" TypeDeclaration? "\$" VarName "in" Expr	-> ForClause
SequenceType	-> TypeDeclaration
(LetClause "return")* TypeswitchExpr	-> LetExpr
"let" TypeDeclaration? "\$" VarName ":@" Expr	-> LetClause
("typeswitch" "(" Expr ")" CaseClause+ "default" "\$"	
VarName "return") * IfExpr	-> TypeswitchExpr
"case" SequenceType "\$" VarName "return" Expr	-> CaseClause
"cast" "as" SequenceType ParenthesizedExpr	-> CastExpr
(ItemType OccurrenceIndicator?) "empty"	-> SequenceType
((("element" "attribute") ElemOrAttrType?)	-> ItemType {prefer}
"processing-instruction" "comment" "item"	-> ItemType
"document" AtomicType "node" "atomic" "value"	-> ItemType
QName	-> AtomicType
(QName (SchemaType SchemaContext?)) SchemaType	-> ElemOrAttrType
"of" "type" QName	-> SchemaType
("if" "(" Expr ")" "then" Expr "else")* ValueExpr	-> IfExpr
ValidateExpr CastExpr Constructor PathExpr	-> ValueExpr
"validate" SchemaContext? "{" Expr "}"	-> ValidateExpr
"in" SchemaGlobalContext ("/" SchemaContextStep)*	-> SchemaContext
QName ("type" QName)	-> SchemaGlobalContext
QName	-> SchemaContextStep
RelativePathExpr	-> PathExpr
StepExpr	-> RelativePathExpr
ForwardStep ReverseStep PrimaryExpr	-> StepExpr
ForwardAxis NodeTest	-> ForwardStep
ReverseAxis NodeTest	-> ReverseStep
XmlComment XmlProcessingInstruction	-> Constructor
ComputedDocumentConstructor	-> Constructor
ComputedElementConstructor	-> Constructor
ComputedAttributeConstructor	-> Constructor
"{" ExprSequence "}"	-> EnclosedExpr

"<!--" Char* "-->"	-> XmlComment
"<?" PITarget Char* ">"	-> XmlProcessingInstruction
"document" "{" ExprSequence "}"	-> ComputedDocumentConstructor
((("element" QName "{") ((("element" "{") Expr "}"{"{"))	
ExprSequence? "}"	-> ComputedElementConstructor
((("attribute" QName "{") ((("attribute" "{") Expr "}"{"{"))	
ExprSequence? "}"	-> ComputedAttributeConstructor
Literal FunctionCall (" \$" VarName)	-> PrimaryExpr
ParenthesizedExpr	-> PrimaryExpr
NumericLiteral StringLiteral	-> Literal
IntegerLiteral DecimalLiteral DoubleLiteral	-> NumericLiteral
(" ExprSequence? ")	-> ParenthesizedExpr
QName "(" ({Expr ", "* }? ")	-> FunctionCall {avoid}
("child" "::") ("descendant" "::")	-> ForwardAxis
("attribute" "::") ("self" "::")	-> ForwardAxis
("descendant-or-self" "::")	-> ForwardAxis
("following-sibling" "::") ("following" "::")	-> ForwardAxis
("namespace" "::")	-> ForwardAxis
("parent" "::") ("ancestor" "::")	-> ReverseAxis
("preceding-sibling" "::")	-> ReverseAxis
("preceding" "::") ("ancestor-or-self" "::")	-> ReverseAxis
NameTest KindTest	-> NodeTest
QName Wildcard	-> NameTest
"*" (NCName ":" "*") ("*" ":" NCName)	-> Wildcard
ProcessingInstructionTest CommentTest	-> KindTest
TextTest AnyKindTest	-> KindTest
"processing-instruction" "(" StringLiteral? ")	-> ProcessingInstructionTest
"comment" "(" ")"	-> CommentTest
"text" "(" ")"	-> TextTest
"node" "(" ")"	-> AnyKindTest

variables

"QBody"	-> QueryBody
"E"[0-9]*	-> Expr
"Es"[0-9]*	-> {Expr " , "}*
"ExprSeq"	-> ExprSequence
"SeqT"[0-9]*	-> SequenceType
"SeqTs"	-> {SequenceType " , "}*
"ITs"[0-9] *	-> {SequenceType " , "}*
"TEExpr"	-> TypeswitchExpr
"UnExpr"	-> UnorderedExpr
"ValExpr"	-> ValueExpr
"For"	-> ForExpr
"Let"	-> LetExpr
"ForCla"	-> ForClause
"NTest"	-> NodeTest
"SchemaT"[0-9]*	-> SchemaType
"Lit"	-> Literal
"FuncCall"	-> FunctionCall
"Constr"	-> Constructor
"UnorExpr"	-> UnorderedExpr
"If"	-> IfExpr
"Cast"	-> CastExpr
"FLEExpr"	-> ForExpr LetExpr
"Parameter"	-> Param

A.4 TypeSystem.sdf

```

module TypeSystem
imports XQueryCoreSyntax
exports
  sorts Value SimpleValue ElementValue AttributeValue DocumentValue TextValue
        NodeValue Item AtomicValue AtomicTypeValue TypeAnnotation ElementName
        AttributeName TypeName Type AtomicTypeName ElementType AttributeType

```

Nillable TypeSpecifier TypeDerivation TypeReference Derivation Mixed
 Definition Substitution SimpleType AttributeAll ElementAll
 ElementContent ElementNoAll ElementBody TFloat TDuration TDateTime
 TTime TDate TGYearMonth TGYear TGMonthDay TGDay TGMonth THexBinary
 TBase64Binary TAnyURI TNOTATION ListType

context-free syntax

Item	-> Value	
Value "," Value	-> Value	
"(")"	-> Value	
error	-> Value	
"(" Value ")"	-> Value {bracket, avoid}	
AtomicValue	-> SimpleValue	
SimpleValue "," SimpleValue	-> SimpleValue	
"(")"	-> SimpleValue	
"element" ElementName TypeAnnotation? "{" Value "}"	-> ElementValue	
"attribute" AttributeName TypeAnnotation? "{" SimpleValue "}"	-> AttributeValue	
"document" "{" Value "}"	-> DocumentValue	
"text" "{" StringLiteral "}"	-> TextValue	
ElementValue AttributeValue DocumentValue TextValue	-> NodeValue	
NodeValue AtomicValue	-> Item	
AtomicTypeValue TypeAnnotation?	-> AtomicValue	
StringLiteral Boolean DecimalLiteral TFloat	-> AtomicTypeValue	
DoubleLiteral TDuration TDateTime TTime	-> AtomicTypeValue	
IntegerLiteral TDate TGYearMonth TGYear TGMonthDay	-> AtomicTypeValue	
TGDay TGMonth THexBinary TBase64Binary	-> AtomicTypeValue	
TAnyURI QName TNOTATION "error"	-> AtomicTypeValue	
("of" "type") TypeName	-> TypeAnnotation	
QName Wildcard	-> ElementName	
QName Wildcard	-> AttributeName	
QName	-> TypeName	
ItemType	-> Type {avoid}	

```

Type OccurrenceIndicator -> Type
Type "&" Type    -> Type {left}
Type "," Type    -> Type {left, avoid}
Type "|" Type    -> Type {left}
 "(" Type ")"    -> Type {bracket}
 "(""")"        -> Type
"none"          -> Type
"error"         -> Type

%seguinte producao incluída para validar equations de Aux-Judgments%
AttributeAll    -> ItemType
ElementContent  -> ItemType
"static error" -> Type

"xs:integer" | "xs:decimal" | "xs:double" | "xs:string" | "xs:boolean" -> Type
"xs:float" | "xs:duration" | "xs:dateTime" | "xs:time" | "xs:date"      -> Type
"xs:gYearMonth" | "xs:gYear" | "xs:gMonthDay" | "xs:gDay" | "xs:gMonth" -> Type
"xs:hexBinary" | "xs:base64Binary" | "xs:anyURI" | "xs:NOTATION"        -> Type
"xs:QName"                                           -> Type

ElementType          -> ItemType
AttributeType         -> ItemType
"document" ("{" Type? "}")? -> ItemType
"text" | "untyped"    -> ItemType
AtomicTypeName        -> ItemType
"error" | "none"      -> ItemType

QName                -> AtomicTypeName
"element" ElementName? Nillable? TypeSpecifier? -> ElementType
"attribute" AttributeName? TypeSpecifier?        -> AttributeType
"nillable"                                         -> Nillable
TypeDerivation | TypeReference                    -> TypeSpecifier
Derivation? Mixed? "{" Type? "}"                 -> TypeDerivation
"of" "type" TypeName                             -> TypeReference
"restricts" TypeName                             -> Derivation
"extends" TypeName                               -> Derivation

```

"mixed"	-> Mixed
("define" "element") ElementName Substitution?	
Nillable? TypeSpecifier	-> Definition
("define" "attribute") AttributeName TypeSpecifier	-> Definition
("define" "type") TypeName TypeDerivation	-> Definition
("substitutes" "for") ElementName	-> Substitution
AtomicTypeName	-> SimpleType
SimpleType OccurrenceIndicator	-> SimpleType
SimpleType " " SimpleType	-> SimpleType
"("")"	-> SimpleType
"none"	-> SimpleType
AttributeType	-> AttributeAll
AttributeType "?"	-> AttributeAll
AttributeAll "&" AttributeAll	-> AttributeAll
"("")"	-> AttributeAll
ElementType	-> ElementAll
ElementType "?"	-> ElementAll
ElementAll "&" ElementAll	-> ElementAll
"("")"	-> ElementAll
"none"	-> ElementAll
ElementType	-> ElementNoAll
"text"	-> ElementNoAll
ElementType OccurrenceIndicator	-> ElementNoAll
ElementAll ", " ElementAll	-> ElementNoAll
ElementAll " " ElementAll	-> ElementNoAll
"("")"	-> ElementNoAll
"ElementContent"	-> ElementContent
AttributeAll ElementContent	-> ElementBody
"xs:float"	-> TFloat
"xs:duration"	-> TDuration
"xs:dateTime"	-> TDateTime
"xs:time"	-> TTime

```

"xs:date"          -> TDate
"xs:gYearMonth"    -> TGYearMonth
"xs:gYear"         -> TGYear
"xs:gMonthDay"     -> TGMonthDay
"xs:gDay"          -> TGDay
"xs:gMonth"        -> TGMonth
"xs:hexBinary"     -> THexBinary
"xs:base64Binary"  -> TBase64Binary
"xs:anyURI"        -> TAnyURI
"xs:NOTATION"      -> TNOTATION
 "[" {Type " ,"}* "]" -> ListType

```

context-free priorities

```

Type OccurrenceIndicator -> Type >
Type " , " Type         -> Type

```

variables

```

"IT"[0-9]*          -> ItemType
"I"[0-9]*           -> Item
"T"[0-9]*           -> Type
"NodeT"             -> Type
"AttrT"             -> Type
"ChildT"            -> Type
"ExtT"              -> Type
"AdjT"              -> Type
"NodeT"             -> Type
"BaseT"             -> Type?
"TOpc"              -> Type?
"CaseT"[0-9]*       -> Type
"CaseTs"[0-9]*      -> {Type " |"}*
"Ts"[0-9]*          -> {Type " ,"}*
"TPlus"             -> {Type " ,"}+

```

```

"TsOr"[0-9]*      -> {Type "|" }*
"Val"[0-9]*       -> Value
"AttrName"[0-9]*  -> AttributeName
"ElemName"[0-9]*  -> ElementName
"BaseElemName"    -> ElementName
"TName"[0-9]*     -> TypeName
"BaseTName"       -> TypeName
"Nil1"           -> Nillable?
"Mix"            -> Mixed?
"BaseMix"        -> Mixed?
"Subst"          -> Substitution?
"Deriv"          -> Derivation?
"TSpec"          -> TypeSpecifier
"TDeriv"         -> TypeDerivation
"AttrAll"[0-9]*   -> AttributeAll
"ElemCont"[0-9]*  -> ElementContent
"Defs"           -> Definition*
"Def"[0-9]*       -> Definition
"AtomicT"[0-9]*   -> AtomicType
"ElemT"          -> ElementType
"AttrT"          -> AttributeType
"SimpleVal"       -> SimpleValue
"AtomicVal"[0-9]* -> AtomicValue
"AnyUri"         -> TAnyURI
"LT"             -> ListType

```

A.5 SubTypes.sdf e SubTypes.asf

```

module SubTypes
imports AuxiliaryJudgments

exports

```

context-free syntax

```
"subtype" "(" Type "," Type "," StatEnvs ")" -> Boolean
"equivalent" "(" Type "," Type "," StatEnvs ")" -> Boolean
"subtype-list" "(" ListType "," ListType "," StatEnvs ")" -> Boolean
"subtype-choose" "(" Type "," "[" {Type "|" }* "]"
                    "," StatEnvs ")" -> Boolean
```

equations

[subt]

```
(Val matches T1, statEnvs) = true,
(Val matches T2, statEnvs) = true
=====
subtype(T1, T2, statEnvs) = true
```

[subt-1]

```
T1 = T2
=====
subtype(T1, T2, statEnvs) = true
```

[subt-2]

```
subtype(T1, T2, statEnvs) = true,
subtype(T2, T3, statEnvs) = true
=====
subtype(T1, T3, statEnvs) = true
```

[subt-3]

```
subtype(T1, T2, statEnvs) = true,
subtype(T2, T3, statEnvs) = true,
subtype(T3, T4, statEnvs) = true,
=====
subtype(T1, T4, statEnvs) = true
```

```
[default-subt] subtype(T1, T2, statEnvs) = false
```

```
[ 0] subtype(IT, xs:anySimpleType, statEnvs) = true
[ 1] subtype(xs:anySimpleType, xs:anyType, statEnvs) = true
[ 2] subtype(xs:duration, xs:anySimpleType, statEnvs) = true
[ 3] subtype(xs:dateTime, xs:anySimpleType, statEnvs) = true
[ 4] subtype(xs:time, xs:anySimpleType, statEnvs) = true
[ 5] subtype(xs:date, xs:anySimpleType, statEnvs) = true
[ 6] subtype(xs:gYearMonth, xs:anySimpleType, statEnvs) = true
[ 7] subtype(xs:gYear, xs:anySimpleType, statEnvs) = true
[ 8] subtype(xs:gMonthDay, xs:anySimpleType, statEnvs) = true
[ 9] subtype(xs:gDay, xs:anySimpleType, statEnvs) = true
[10] subtype(xs:gMonth, xs:anySimpleType, statEnvs) = true
[11] subtype(xs:boolean, xs:anySimpleType, statEnvs) = true
[12] subtype(xs:base64Binary, xs:anySimpleType, statEnvs) = true
[13] subtype(xs:float, xs:anySimpleType, statEnvs) = true
[14] subtype(xs:double, xs:anySimpleType, statEnvs) = true
[15] subtype(xs:anyURI, xs:anySimpleType, statEnvs) = true
[16] subtype(xs:QName, xs:anySimpleType, statEnvs) = true
[17] subtype(xs:NOTATION, xs:anySimpleType, statEnvs) = true
[18] subtype(xs:string, xs:anySimpleType, statEnvs) = true
[19] subtype(xs:decimal, xs:anySimpleType, statEnvs) = true
[20] subtype(xs:normalizedString, xs:string, statEnvs) = true
[21] subtype(xs:integer, xs:decimal, statEnvs) = true
[22] subtype(xs:duration, xs:anySimpleType, statEnvs) = true
```

```
[eqv1-1]
```

```
    subtype(T1, T2, statEnvs) = true,
```

```
    subtype(T2, T1, statEnvs) = true
```

```
=====
```

```
    equivalent(T1, T2, statEnvs) = true
```

```

[list-1] subtype-list([], [], statEnvs) = true

[list-2] subtype-list([T], [T], statEnvs) = subtype(T, T, statEnvs)

[list-3]
    subtype(T1, T2, statEnvs) = true,
    subtype-list([Ts1], [Ts2], statEnvs) = true
    =====
    subtype-list([T1, Ts1], [T2, Ts2], statEnvs) = true

[default-list] subtype-list([Ts1], [Ts2], statEnvs) = false

[choo-1] subtype-choose(T, [], statEnvs) = false

[choo-2]
    subtype(T, T1, statEnvs) = true,
    subtype-choose(T, [TsOr], statEnvs) = true
    =====
    subtype-choose(T, [T1 | TsOr], statEnvs) = true

```

A.6 PrimeTypes.sdf e PrimeTypes.asf

```

module PrimeTypes
imports SubTypes
exports
    sorts PrimeType OccurrIndicator

context-free syntax
    Type | ItemType*                -> PrimeType
    PrimeType "|" PrimeType         -> PrimeType
    "prime" "(" Type ")"            -> Type
    Type "." OccurrIndicator        -> Type

```

```

"+" | "*" | "?" | "0" | "1"          -> OccurrIndicator
ItemType                             -> OccurrIndicator
OccurrIndicator "|" OccurrIndicator -> OccurrIndicator
OccurrIndicator "," OccurrIndicator -> OccurrIndicator
OccurrIndicator "." OccurrIndicator -> OccurrIndicator
"quantifier" "(" Type ")"            -> OccurrIndicator

```

variables

```

"OccurrInd"[0-9]* -> OccurrIndicator

```

equations

```

[prim-1] prime(IT) = IT
[prim-2] prime(()) = none
[prim-3] prime(none) = none
[prim-4] prime(T1, T2) = prime(T1) | prime(T2)
[prim-5] prime(T1 & T2) = prime(T1) | prime(T2)
[prim-6] prime(T1 | T2) = prime(T1) | prime(T2)
[prim-7] prime(T?) = prime(T)
[prim-8] prime(T*) = prime(T)
[prim-9] prime(T+) = prime(T)
[type-1] T.0 = () | ()
[type-2] T.1 = T
[type-3] T.? = T?
[type-4] T.+ = T+
[type-5] T.* = T*
[qunt-1] quantifier(IT) = 1
[qunt-2] quantifier(()) = 0
[qunt-3] quantifier(none) = 0
[qunt-4] quantifier(T1, T2) = quantifier(T1), quantifier(T2)
[qunt-5] quantifier(T1 & T2) = quantifier(T1), quantifier(T2)
[qunt-6] quantifier(T1 | T2) = quantifier(T1) | quantifier(T2)
[qunt-7] quantifier(T?) = quantifier(T).?

```

```

[quint-8] quantifier(T*) = quantifier(T).*
[quint-9] quantifier(T+) = quantifier(T).+
%[pr-qt] subtype(T, prime(T).quantifier(T), statEnvs) = true%
[sum-1] 0, OccurrInd2 = OccurrInd2
[sum-2] OccurrInd1, 0 = OccurrInd1
[sum-3] +, OccurrInd2 = +
[sum-4] OccurrInd1, + = +

[sum-5]
    OccurrInd2 != 0
    =====
    1, OccurrInd2 = +

[sum-6]
    OccurrInd1 != 0,
    =====
    OccurrInd1 , 1 = +

[sum-7] ?, ? = *
[sum-8] *, * = *
[sum-9] ?, * = *
[sum-10] *, ? = *
[chce-1] * | OccurrInd2 = *
[chce-2] OccurrInd1 | * = *

[chce-3]
    OccurrInd = OccurrInd
    =====
    OccurrInd | OccurrInd = OccurrInd

[chce-4] 1 | + = +
[chce-5] + | 1 = +

```

[chce-6] $0 \mid 1 = ?$
 [chce-7] $1 \mid 0 = ?$
 [chce-8] $0 \mid ? = ?$
 [chce-9] $? \mid 0 = ?$
 [chce-10] $? \mid 1 = ?$
 [chce-11] $1 \mid ? = ?$
 [chce-12] $0 \mid + = *$
 [chce-13] $+ \mid 0 = *$
 [chce-14] $? \mid + = *$
 [chce-15] $+ \mid ? = *$
 [prod-1] $0 \cdot \text{OccurrInd2} = 0$
 [prod-2] $\text{OccurrInd1} \cdot 0 = 0$

[prod-3]
 $\text{OccurrInd} = \text{OccurrInd}$
 =====
 $\text{OccurrInd} \cdot \text{OccurrInd} = \text{OccurrInd}$

[prod-4]
 $\text{OccurrInd2} \neq 0$
 =====
 $* \cdot \text{OccurrInd2} = *$

[prod-5]
 $\text{OccurrInd1} \neq 0$
 =====
 $\text{OccurrInd1} \cdot * = *$

[prod-6] $1 \cdot + = +$
 [prod-7] $+ \cdot 1 = +$
 [prod-8] $1 \cdot ? = ?$
 [prod-9] $? \cdot 1 = ?$

[prod-10] ? . + = *

[prod-11] + . ? = *

A.7 TypeInteger.sdf e TypeInteger.asf

```
module TypeInteger
imports TypeBoolean
exports
  sorts PICO-NAT-CON PICO-NAT PICO-INT

lexical syntax
  [0-9]+ -> PICO-NAT-CON

context-free syntax
  PICO-NAT-CON          -> PICO-NAT
  PICO-NAT "-/" PICO-NAT -> PICO-NAT {left}
  "(" PICO-NAT ")"      -> PICO-NAT {bracket}
  PICO-NAT              -> PICO-INT
  "+" PICO-NAT          -> PICO-INT
  "-" PICO-NAT          -> PICO-INT
  PICO-INT "+" PICO-INT -> PICO-INT {left}
  PICO-INT "-" PICO-INT -> PICO-INT {left}
  PICO-INT "*" PICO-INT -> PICO-INT {left}
  PICO-INT ">" PICO-INT   -> Boolean
  PICO-INT ">=" PICO-INT  -> Boolean
  PICO-INT "<" PICO-INT   -> Boolean
  PICO-INT "<=" PICO-INT  -> Boolean
  "(" PICO-INT ")"      -> PICO-INT {bracket}

variables
  "Nat" [0-9\']* -> PICO-NAT
  "Nat-con" [0-9\']* -> PICO-NAT-CON
```

"Int" [0-9\']* -> PICO-INT

context-free priorities

PICO-INT "*" PICO-INT -> PICO-INT >

{left: PICO-INT "+" PICO-INT -> PICO-INT

PICO-INT "-" PICO-INT -> PICO-INT}

hiddens

context-free syntax

gt "(" PICO-NAT "," PICO-NAT ")" -> Boolean

PICO-NAT "-/" PICO-NAT -> PICO-NAT

PICO-NAT ">-" PICO-NAT -> PICO-NAT

variables

"n"[0-9]* -> PICO-NAT-CON

"c"[0-9]* -> CHAR

"x"[0-9]* -> CHAR*

"y"[0-9]* -> CHAR+

equations

[1] pico-nat-con("0" y) = pico-nat-con(y)

%% -- addition --

[2] 0 + Nat = Nat

[3] Nat + 0 = Nat

[4] 1 + 1 = 2 [5] 1 + 2 = 3 [6] 1 + 3 = 4

[7] 1 + 4 = 5 [8] 1 + 5 = 6 [9] 1 + 6 = 7

[10] 1 + 7 = 8 [11] 1 + 8 = 9 [12] 1 + 9 = 10

[13] 2 + 1 = 3 [14] 2 + 2 = 4 [15] 2 + 3 = 5

[16] 2 + 4 = 6 [17] 2 + 5 = 7 [18] 2 + 6 = 8

[19] 2 + 7 = 9 [20] 2 + 8 = 10 [21] 2 + 9 = 11

[22] 3 + 1 = 4 [23] 3 + 2 = 5 [24] 3 + 3 = 6

[25] 3 + 4 = 7 [26] 3 + 5 = 8 [27] 3 + 6 = 9

[28] $3 + 7 = 10$	[29] $3 + 8 = 11$	[30] $3 + 9 = 12$
[31] $4 + 1 = 5$	[32] $4 + 2 = 6$	[33] $4 + 3 = 7$
[34] $4 + 4 = 8$	[35] $4 + 5 = 9$	[36] $4 + 6 = 10$
[37] $4 + 7 = 11$	[38] $4 + 8 = 12$	[39] $4 + 9 = 13$
[40] $5 + 1 = 6$	[41] $5 + 2 = 7$	[42] $5 + 3 = 8$
[43] $5 + 4 = 9$	[44] $5 + 5 = 10$	[45] $5 + 6 = 11$
[46] $5 + 7 = 12$	[47] $5 + 8 = 13$	[48] $5 + 9 = 14$
[49] $6 + 1 = 7$	[50] $6 + 2 = 8$	[51] $6 + 3 = 9$
[52] $6 + 4 = 10$	[53] $6 + 5 = 11$	[54] $6 + 6 = 12$
[55] $6 + 7 = 13$	[56] $6 + 8 = 14$	[57] $6 + 9 = 15$
[58] $7 + 1 = 8$	[59] $7 + 2 = 9$	[60] $7 + 3 = 10$
[61] $7 + 4 = 11$	[62] $7 + 5 = 12$	[63] $7 + 6 = 13$
[64] $7 + 7 = 14$	[65] $7 + 8 = 15$	[66] $7 + 9 = 16$
[67] $8 + 1 = 9$	[68] $8 + 2 = 10$	[69] $8 + 3 = 11$
[70] $8 + 4 = 12$	[71] $8 + 5 = 13$	[72] $8 + 6 = 14$
[73] $8 + 7 = 15$	[74] $8 + 8 = 16$	[75] $8 + 9 = 17$
[76] $9 + 1 = 10$	[77] $9 + 2 = 11$	[78] $9 + 3 = 12$
[79] $9 + 4 = 13$	[80] $9 + 5 = 14$	[81] $9 + 6 = 15$
[82] $9 + 7 = 16$	[83] $9 + 8 = 17$	[84] $9 + 9 = 18$

[85a]

```
pico-nat-con(c1) + pico-nat-con(c2) = pico-nat-con(x c),
pico-nat-con (y1) + pico-nat-con(y2) + pico-nat-con("0" x) = pico-nat-con(y)
=====
pico-nat-con(y1 c1) + pico-nat-con(y2 c2) = pico-nat-con(y c)
```

[85b]

```
pico-nat-con(c1) + pico-nat-con(c2) = pico-nat-con(x c),
pico-nat-con(y2) + pico-nat-con("0" x) = pico-nat-con(y)
=====
pico-nat-con(c1) + pico-nat-con(y2 c2) = pico-nat-con(y c)
```

[85c]

```
pico-nat-con(c1) + pico-nat-con(c2) = pico-nat-con(x c),  
pico-nat-con (y1) + pico-nat-con("0" x) = pico-nat-con(y)  
=====
```

```
pico-nat-con(y1 c1) + pico-nat-con(c2) = pico-nat-con(y c)
```

[86] pico-nat-con(c) -// pico-nat-con(c) = 0

[87] pico-nat-con(c) -// 0 = pico-nat-con(c)

[88] 2 -// 1 = 1

[89] 3 -// 1 = 2 [90] 3 -// 2 = 1

[91] 4 -// 1 = 3 [92] 4 -// 2 = 2 [93] 4 -// 3 = 1

[94] 5 -// 1 = 4 [95] 5 -// 2 = 3 [96] 5 -// 3 = 2

[97] 5 -// 4 = 1

[98] 6 -// 1 = 5 [99] 6 -// 2 = 4 [100] 6 -// 3 = 3

[101] 6 -// 4 = 2 [102] 6 -// 5 = 1

[103] 7 -// 1 = 6 [104] 7 -// 2 = 5 [105] 7 -// 3 = 4

[106] 7 -// 4 = 3 [107] 7 -// 5 = 2 [108] 7 -// 6 = 1

[109] 8 -// 1 = 7 [110] 8 -// 2 = 6 [111] 8 -// 3 = 5

[112] 8 -// 4 = 4 [113] 8 -// 5 = 3 [114] 8 -// 6 = 2

[115] 8 -// 7 = 1

[116] 9 -// 1 = 8 [117] 9 -// 2 = 7 [118] 9 -// 3 = 6

[119] 9 -// 4 = 5 [120] 9 -// 5 = 4 [121] 9 -// 6 = 3

[122] 9 -// 7 = 2 [123] 9 -// 8 = 1

[124] 10 -// 1 = 9 [125] 10 -// 2 = 8 [126] 10 -// 3 = 7

[127] 10 -// 4 = 6 [128] 10 -// 5 = 5 [129] 10 -// 6 = 4

[130] 10 -// 7 = 3 [131] 10 -// 8 = 2 [132] 10 -// 9 = 1

%% -- auxiliary function gt

[133]

```
pico-nat-con(c1) -// pico-nat-con(c2) = pico-nat-con(c),  
pico-nat-con(c) != 0
```

```

=====
gt(pico-nat-con(c1), pico-nat-con(c2)) = true

[134] gt(pico-nat-con(y1 c1), pico-nat-con(c2)) = true
[135] gt(pico-nat-con(y c1), pico-nat-con(y c2)) = gt(pico-nat-con(c1), pico-nat-con(c2))
[136]
gt(pico-nat-con(y1), pico-nat-con(y2)) = true
=====
gt(pico-nat-con(y1 c1), pico-nat-con(y2 c2)) = true

%% -- auxiliary function >- , partial subtraction

[137] Nat >- 0 = Nat

[138] pico-nat-con(c1) >- pico-nat-con(c2) = pico-nat-con(c1) -// pico-nat-con(c2)

[139a]
pico-nat-con(c1) -// pico-nat-con(c2) = pico-nat-con(c3),
pico-nat-con (y1) >- pico-nat-con("0" x2) = pico-nat-con(y)
=====
pico-nat-con(y1 c1) >- pico-nat-con(x2 c2) = pico-nat-con(y c3)

[139b]
pico-nat-con(c2) -// pico-nat-con(c1) = pico-nat-con(c3),
10 -// pico-nat-con(c3) = pico-nat-con(c),
pico-nat-con ("0" x2) + 1 = Nat,
pico-nat-con(y1) >- Nat = pico-nat-con(y)
=====
pico-nat-con(y1 c1) >- pico-nat-con(x2 c2) = pico-nat-con(y c)

%% -- cut off subtraction -/

[140] gt(Nat1, Nat2) = true === Nat1 -/ Nat2 = Nat1 >- Nat2

```

```

[141] gt(Nat1, Nat2) != true == Nat1 -/ Nat2 = 0

%% -- subtraction of naturals

[142] gt(Nat1, Nat2) = true == Nat1 - Nat2 = Nat1 >- Nat2

[143] gt(Nat1, Nat2) != true == Nat1 - Nat2 = -(Nat2 >- Nat1)

%% -- multiplication of naturals --
[144] Nat * 0 = 0
[145] Nat * 1 = Nat
[146]
    gt(pico-nat-con(c), 1) = true
    =====
    Nat * pico-nat-con(c) = Nat + Nat * (pico-nat-con(c) - 1)

[147]
    pico-nat-con(y1) * pico-nat-con(y2) = pico-nat-con(y)
    =====
    pico-nat-con(y1) * pico-nat-con(y2 c) =
        pico-nat-con(y "0") + pico-nat-con(y1) * pico-nat-con(c)

%% -- addition, subtraction, and multiplication of integers
[148] +Nat = Nat

[149] Nat1 + -Nat2 = Nat1 - Nat2
[150] -Nat1 + Nat2 = Nat2 - Nat1
[151] Nat1 + Nat2 = Nat == -Nat1 + -Nat2 = -Nat

[152] Nat1 - -Nat2 = Nat1 + Nat2
[153] Nat1 + Nat2 = Nat == -Nat1 - Nat2 = -Nat
[154] -Nat1 - -Nat2 = Nat2 - Nat1

```

```

[155] Nat1 * Nat2 = Nat  === Nat1 * -Nat2 = -Nat
[156] Nat1 * Nat2 = Nat  === -Nat1 * Nat2 = -Nat
[157] -Nat1 * -Nat2 = Nat1 * Nat2

%% -- relational operators
[158] gt(Nat1, Nat2) = true === Nat1 > Nat2 = true
[159] gt(Nat1, Nat2) != true === Nat1 > Nat2 = false
[160] Nat1 > -Nat2 = true
[161] -Nat1 > Nat2 = false
[162] -Nat1 > -Nat2 = Nat2 > Nat1

[163] Int1 != Int2 === Int1 >= Int2 = Int1 > Int2
[164] Int >= Int = true

[165] Int1 < Int2 = not(Int1 >= Int2)
[166] Int1 <= Int2 = not(Int1 > Int2)

%% -- no minus zero
[167] - 0 = 0

```

A.8 TypeBoolean.sdf e TypeBoolean.asf

```

module TypeBoolean
imports Layout
exports
  sorts Boolean

context-free syntax
  "true" | "false" | "0" | "1" -> Boolean
  Boolean "|" Boolean           -> Boolean {left}
  Boolean "&" Boolean            -> Boolean {left}

```

```

"not" "(" Boolean ")"      -> Boolean
"(" Boolean ")"            -> Boolean {bracket}

variables
  "BoolVal" -> Boolean

context-free priorities
  Boolean "&" Boolean -> Boolean >
  Boolean "|" Boolean -> Boolean

equations

[bool-1] true | BoolVal = true
[bool-2] false | BoolVal = BoolVal
[bool-3] true & BoolVal = BoolVal
[bool-4] false & BoolVal = false
[bool-5] not(true) = false
[bool-6] not(false) = true
[bool-1a] 1 | BoolVal = 1
[bool-2a] 0 | BoolVal = BoolVal
[bool-3a] 1 & BoolVal = BoolVal
[bool-4a] 0 & BoolVal = 0
[bool-5a] not(1) = 0
[bool-6a] not(0) = 1

```

A.9 StatEnvironment.sdf e StatEnvironment.asf

```

module StatEnvironment
  imports TypeSystem
  exports
    sorts StatEnvs ListOfPairs Pair Key Arity Contents QNameValue

```

context-free syntax

```

ListOfPairs "," ListOfPairs "," ListOfPairs "," ListOfPairs ","
ListOfPairs "," ListOfPairs "," ListOfPairs "," ListOfPairs      -> StatEnvs
StatEnvs "[" Contents "]"                                         -> StatEnvs
"[" Pair* "]"                                                     -> ListOfPairs
"(" Key "-->" Contents ")"                                         -> Pair
NCName | TypeName | ElementName | AttributeName | (" $" VarName) -> Key
"element" | "function"                                           -> Key
((" QName ") | Prefix | LocalPart                                -> Key
PICO-NAT-CON                                                      -> Arity
StringLiteral | Definition | Type+                                -> Contents
"define" "function" QName "(" {SequenceType ","}* ")" "returns"
                                                                    SequenceType -> Contents
{Type ","}*                                                       -> SequenceType
"namespace"                "(" StatEnvs ")"                        -> ListOfPairs
"default-namespace"        "(" StatEnvs ")"                        -> ListOfPairs
"type-defn"                "(" StatEnvs ")"                        -> ListOfPairs
"elem-decl"                "(" StatEnvs ")"                        -> ListOfPairs
"attr-decl"                "(" StatEnvs ")"                        -> ListOfPairs
"var-type"                 "(" StatEnvs ")"                        -> ListOfPairs
"func-type"                "(" StatEnvs ")"                        -> ListOfPairs
"base-uri"                 "(" StatEnvs ")"                        -> ListOfPairs
"insert" Pair "in" ("NS"|"DNS"|"TD"|"ED"|"AD"|"VT"|"FT"|"BU")
                                                                    "of" StatEnvs -> StatEnvs
"lookup" "(" Key "," ListOfPairs ")"                              -> Contents
"expand" "(" QName "," StatEnvs ")"                                -> QNameValue
"qname" "(" StringLiteral "," NCName ")"                           -> QNameValue
"qname" "(" NCName ")"                                             -> QNameValue

```

variables

```

"statEnvs"[0-9\']* -> StatEnvs
"LP"[0-9\']*       -> ListOfPairs

```

```

"K"[0-9\']*      -> Key
"C"[0-9\']*      -> Contents
"P"[0-9\']*      -> Pair
"Ps"[0-9\']*     -> Pair*
"n"[0-9\']*      -> Arity

```

equations

```

[stat-1] namespace(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP1
[stat-2] default-namespace(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP2
[stat-3] type-defn(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP3
[stat-4] elem-decl(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP4
[stat-5] attr-decl(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP5
[stat-6] var-type(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP6
[stat-7] func-type(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP7
[stat-8] base-uri(LP1, LP2, LP3, LP4, LP5, LP6, LP7, LP8) = LP8

```

```

[look-a] lookup(K, []) = error

```

```

[look-b]
    lookup(K, [(K --> C) Ps]) = C

```

```

[look-c]
    K != K1,
    lookup(K, [Ps]) = C2
    =====
    lookup(K, [(K1 --> C1) Ps]) = C2

```

```

[insert-1]
    insert P in NS of [Ps], LP2, LP3, LP4, LP5, LP6, LP7, LP8
    = [P Ps], LP2, LP3, LP4, LP5, LP6, LP7, LP8

```

```

[insert-2]
    insert P in DNS of LP1, [Ps], LP3, LP4, LP5, LP6, LP7, LP8

```

```

= LP1, [P Ps], LP3, LP4, LP5, LP6, LP7, LP8

[insert-3]
insert P in TD of LP1, LP2, [Ps], LP4, LP5, LP6, LP7, LP8
= LP1, LP2, [P Ps], LP4, LP5, LP6, LP7, LP8

[insert-4]
insert P in ED of LP1, LP2, LP3, [Ps], LP5, LP6, LP7, LP8
= LP1, LP2, LP3, [P Ps], LP5, LP6, LP7, LP8

[insert-5]
insert P in AD of LP1, LP2, LP3, LP4, [Ps], LP6, LP7, LP8
= LP1, LP2, LP3, LP4, [P Ps], LP6, LP7, LP8

[insert-6]
insert P in VT of LP1, LP2, LP3, LP4, LP5, [Ps], LP7, LP8
= LP1, LP2, LP3, LP4, LP5, [P Ps], LP7, LP8

[insert-7]
insert P in FT of LP1, LP2, LP3, LP4, LP5, LP6, [Ps], LP8
= LP1, LP2, LP3, LP4, LP5, LP6, [P Ps], LP8

[insert-8]
insert P in BU of LP1, LP2, LP3, LP4, LP5, LP6, LP7, [Ps]
= LP1, LP2, LP3, LP4, LP5, LP6, LP7, [P Ps]

[expand-1]
QN = NCN1:NCN2,
lookup(NCN1, namespace(statEnvs)) = Uri
=====
expand(QN, statEnvs) = qname(Uri, NCN2)

[expand-2]
QN = NCN,
lookup(element, default-namespace(statEnvs)) = Uri
=====
expand(NCN, statEnvs) = qname(Uri, NCN)

```

[expand-3]

```
lookup(function, default-namespace(statEnvs)) = StringLit
=====
expand(FunctionName, statEnvs) = qname(StringLit, FunctionName)
```

A.10 DynEnvironment.sdf e DynEnvironment.asf

```
module DynEnvironment
imports TypeSystem %Auxiliary%
exports
  sorts DynEnvs ListOfP PairDyn KeyDyn ArityDyn ContentsDyn ValueList VarList

context-free syntax
  ListOfP "," ListOfP                                -> DynEnvs
  "[" PairDyn* "]"                                    -> ListOfP
  "(" KeyDyn "-->" ContentsDyn ")"                    -> PairDyn
  QName | VarName                                     -> KeyDyn
  "(" QName "," ArityDyn ")"                           -> KeyDyn
  "integer"                                            -> ArityDyn
  ValueList                                            -> ContentsDyn
  Value                                                -> ContentsDyn
  "(" Expr "," VarList ")"                             -> ValueList
  {VarName ","}*                                       -> VarList
  "func-defn" "(" DynEnvs ")"                          -> ListOfP
  "var-value" "(" DynEnvs ")"                          -> ListOfP
  "insert" PairDyn "in" ("FD"|"VV") "of" DynEnvs -> DynEnvs
  "lookupDyn" "(" KeyDyn "," ListOfP ")"               -> ContentsDyn

variables
  "dynEnvs"[0-9]*   -> DynEnvs
  "ListP"[0-9]*     -> ListOfP
  "KD" [0-9]*       -> KeyDyn
```

```

"CD" [0-9]*      -> ContentsDyn
"PDs"[0-9]*      -> PairDyn*
"PD"[0-9]*       -> PairDyn

equations

[dyn-1] func-defn(ListP1, ListP2) = ListP1
[dyn-2] var-value(ListP1, ListP2) = ListP2

[look-a] lookupDyn(KD, []) = error

[look-b] lookupDyn(KD, [(KD--> CD) PDs]) = CD

[look-c]
    KD != KD1,
    lookupDyn(KD, [PDs]) = CD2,
    =====
    lookupDyn(KD, [(KD1 --> CD1) PDs]) = CD2

[ins-1] insert PD in FD of [PDs], ListP2 = [PD PDs], ListP2

[ins-2] insert PD in VV of ListP1, [PDs] = ListP1, [PD PDs]

```

A.11 FunctionsOnQNames.sdf e FunctionsOnQNames.asf

```

module FunctionsOnQNames
imports TerminalsSyntax
exports
    context-free syntax
    ("xf" ":" "get-namespace-uri") "(" QName ")" -> NCName
    ("xf" ":" "get-local-name")      "(" QName ")" -> NCName

equations

```

```
[names-1]
  QN = NCN1:NCN2
  =====
  xf:get-namespace-uri(QN) = NCN1
```

```
[local-1]
  QN = NCN1:NCN2
  =====
  xf:get-local-name(QN) = NCN2
```

A.12 CastingTable.sdf e CastingTable.asf

```
module CastingTable
imports TypeSystem
exports
  sorts Set

context-free syntax
  Type "cast" "allowed" Type -> Set
  "Y" | "M" | "N"           -> Set %Yes Maybe No%

equations

[1]
  T = T1
  =====
  T cast allowed T1 = Y

[2]  xs:string cast allowed xs:float      = M
[3]  xs:string cast allowed xs:double    = M
[4]  xs:string cast allowed xs:decimal   = M
```

[5]	xs:string cast allowed xs:duration	= M
[6]	xs:string cast allowed xs:dateTime	= M
[7]	xs:string cast allowed xs:time	= M
[8]	xs:string cast allowed xs:date	= M
[9]	xs:string cast allowed xs:gYearMonth	= M
[10]	xs:string cast allowed xs:gYear	= M
[11]	xs:string cast allowed xs:gMonthDay	= M
[12]	xs:string cast allowed xs:gDay	= M
[13]	xs:string cast allowed xs:gMonth	= M
[14]	xs:string cast allowed xs:boolean	= M
[15]	xs:string cast allowed xs:base64Binary	= N
[16]	xs:string cast allowed xs:hexBinary	= N
[17]	xs:string cast allowed xs:anyURI	= M
[18]	xs:string cast allowed xs:QN	= N
[19]	xs:string cast allowed xs:NOTATION	= M
[20]	xs:float cast allowed xs:string	= Y
[21]	xs:float cast allowed xs:double	= Y
[22]	xs:float cast allowed xs:decimal	= M
[23]	xs:float cast allowed xs:duration	= N
[24]	xs:float cast allowed xs:dateTime	= N
[25]	xs:float cast allowed xs:time	= N
[26]	xs:float cast allowed xs:date	= N
[27]	xs:float cast allowed xs:gYearMonth	= N
[28]	xs:float cast allowed xs:gYear	= N
[29]	xs:float cast allowed xs:gMonthDay	= N
[30]	xs:float cast allowed xs:gDay	= N
[31]	xs:float cast allowed xs:gMonth	= N
[32]	xs:float cast allowed xs:boolean	= M
[33]	xs:float cast allowed xs:base64Binary	= N
[34]	xs:float cast allowed xs:hexBinary	= N
[35]	xs:float cast allowed xs:anyURI	= N

[36]	xs:float cast allowed xs:QN	= N
[37]	xs:float cast allowed xs:NOTATION	= N
[38]	xs:double cast allowed xs:string	= Y
[39]	xs:double cast allowed xs:float	= Y
[40]	xs:double cast allowed xs:decimal	= M
[41]	xs:double cast allowed xs:duration	= N
[42]	xs:double cast allowed xs:dateTime	= N
[43]	xs:double cast allowed xs:time	= N
[44]	xs:double cast allowed xs:date	= N
[45]	xs:double cast allowed xs:gYearMonth	= N
[46]	xs:double cast allowed xs:gYear	= N
[47]	xs:double cast allowed xs:gMonthDay	= N
[48]	xs:double cast allowed xs:gDay	= N
[49]	xs:double cast allowed xs:gMonth	= N
[50]	xs:double cast allowed xs:boolean	= M
[51]	xs:double cast allowed xs:base64Binary	= N
[52]	xs:double cast allowed xs:hexBinary	= N
[53]	xs:double cast allowed xs:anyURI	= N
[54]	xs:double cast allowed xs:QN	= N
[55]	xs:double cast allowed xs:NOTATION	= N
[56]	xs:decimal cast allowed xs:string	= Y
[57]	xs:decimal cast allowed xs:float	= Y
[58]	xs:decimal cast allowed xs:double	= Y
[59]	xs:decimal cast allowed xs:duration	= N
[60]	xs:decimal cast allowed xs:dateTime	= N
[61]	xs:decimal cast allowed xs:time	= N
[62]	xs:decimal cast allowed xs:date	= N
[63]	xs:decimal cast allowed xs:gYearMonth	= N
[64]	xs:decimal cast allowed xs:gYear	= N
[65]	xs:decimal cast allowed xs:gMonthDay	= N

[66] xs:decimal cast allowed xs:gDay = N
 [67] xs:decimal cast allowed xs:gMonth = N
 [68] xs:decimal cast allowed xs:boolean = M
 [69] xs:decimal cast allowed xs:base64Binary = N
 [70] xs:decimal cast allowed xs:hexBinary = N
 [71] xs:decimal cast allowed xs:anyURI = N
 [72] xs:decimal cast allowed xs:QN = N
 [73] xs:decimal cast allowed xs:NOTATION = N

[74] xs:duration cast allowed xs:string = Y
 [75] xs:duration cast allowed xs:float = N
 [76] xs:duration cast allowed xs:double = N
 [77] xs:duration cast allowed xs:decimal = N
 [78] xs:duration cast allowed xs:dateTime = N
 [79] xs:duration cast allowed xs:time = N
 [80] xs:duration cast allowed xs:date = N
 [81] xs:duration cast allowed xs:gYearMonth = N
 [82] xs:duration cast allowed xs:gYear = N
 [83] xs:duration cast allowed xs:gMonthDay = N
 [84] xs:duration cast allowed xs:gDay = N
 [85] xs:duration cast allowed xs:gMonth = N
 [86] xs:duration cast allowed xs:boolean = N
 [87] xs:duration cast allowed xs:base64Binary = N
 [88] xs:duration cast allowed xs:hexBinary = N
 [89] xs:duration cast allowed xs:anyURI = N
 [90] xs:duration cast allowed xs:QN = N
 [91] xs:duration cast allowed xs:NOTATION = N

[92] xs:dateTime cast allowed xs:string = Y
 [93] xs:dateTime cast allowed xs:float = N
 [94] xs:dateTime cast allowed xs:double = N
 [95] xs:dateTime cast allowed xs:decimal = N

[96]	xs:dateTime cast allowed xs:duration	= N
[97]	xs:dateTime cast allowed xs:time	= N
[98]	xs:dateTime cast allowed xs:date	= N
[99]	xs:dateTime cast allowed xs:gYearMonth	= N
[100]	xs:dateTime cast allowed xs:gYear	= N
[101]	xs:dateTime cast allowed xs:gMonthDay	= N
[102]	xs:dateTime cast allowed xs:gDay	= N
[103]	xs:dateTime cast allowed xs:gMonth	= N
[104]	xs:dateTime cast allowed xs:boolean	= N
[105]	xs:dateTime cast allowed xs:base64Binary	= N
[106]	xs:dateTime cast allowed xs:hexBinary	= N
[107]	xs:dateTime cast allowed xs:anyURI	= N
[108]	xs:dateTime cast allowed xs:QN	= N
[109]	xs:dateTime cast allowed xs:NOTATION	= N
[110]	xs:time cast allowed xs:string	= Y
[111]	xs:time cast allowed xs:float	= N
[112]	xs:time cast allowed xs:double	= N
[113]	xs:time cast allowed xs:decimal	= N
[114]	xs:time cast allowed xs:duration	= N
[115]	xs:time cast allowed xs:dateTime	= N
[116]	xs:time cast allowed xs:date	= N
[117]	xs:time cast allowed xs:gYearMonth	= N
[118]	xs:time cast allowed xs:gYear	= N
[119]	xs:time cast allowed xs:gMonthDay	= N
[120]	xs:time cast allowed xs:gDay	= N
[121]	xs:time cast allowed xs:gMonth	= N
[122]	xs:time cast allowed xs:boolean	= N
[123]	xs:time cast allowed xs:base64Binary	= N
[124]	xs:time cast allowed xs:hexBinary	= N
[125]	xs:time cast allowed xs:anyURI	= N
[126]	xs:time cast allowed xs:QN	= N

[127] xs:time cast allowed xs:NOTATION = N

[128] xs:date cast allowed xs:string = Y

[129] xs:date cast allowed xs:float = N

[130] xs:date cast allowed xs:double = N

[131] xs:date cast allowed xs:decimal = N

[132] xs:date cast allowed xs:duration = N

[133] xs:date cast allowed xs:dateTime = N

[134] xs:date cast allowed xs:time = N

[135] xs:date cast allowed xs:gYearMonth = Y

[136] xs:date cast allowed xs:gYear = N

[137] xs:date cast allowed xs:gMonthDay = N

[138] xs:date cast allowed xs:gDay = N

[139] xs:date cast allowed xs:gMonth = N

[140] xs:date cast allowed xs:boolean = N

[141] xs:date cast allowed xs:base64Binary = N

[142] xs:date cast allowed xs:hexBinary = N

[143] xs:date cast allowed xs:anyURI = N

[144] xs:date cast allowed xs:QN = N

[145] xs:date cast allowed xs:NOTATION = N

[146] xs:gYearMonth cast allowed xs:string = Y

[147] xs:gYearMonth cast allowed xs:float = N

[148] xs:gYearMonth cast allowed xs:double = N

[149] xs:gYearMonth cast allowed xs:decimal = N

[150] xs:gYearMonth cast allowed xs:duration = N

[151] xs:gYearMonth cast allowed xs:dateTime = N

[152] xs:gYearMonth cast allowed xs:time = N

[153] xs:gYearMonth cast allowed xs:date = N

[154] xs:gYearMonth cast allowed xs:gYear = N

[155] xs:gYearMonth cast allowed xs:gMonthDay = N

[156] xs:gYearMonth cast allowed xs:gDay = N

[157] xs:gYearMonth cast allowed xs:gMonth = N
 [158] xs:gYearMonth cast allowed xs:boolean = N
 [159] xs:gYearMonth cast allowed xs:base64Binary = N
 [160] xs:gYearMonth cast allowed xs:hexBinary = N
 [161] xs:gYearMonth cast allowed xs:anyURI = N
 [162] xs:gYearMonth cast allowed xs:QN = N
 [163] xs:gYearMonth cast allowed xs:NOTATION = N

[164] xs:gYear cast allowed xs:string = Y
 [165] xs:gYear cast allowed xs:float = N
 [166] xs:gYear cast allowed xs:double = N
 [167] xs:gYear cast allowed xs:decimal = N
 [167] xs:gYear cast allowed xs:duration = N
 [169] xs:gYear cast allowed xs:dateTime = N
 [170] xs:gYear cast allowed xs:time = N
 [171] xs:gYear cast allowed xs:date = N
 [172] xs:gYear cast allowed xs:gYearMonth = N
 [173] xs:gYear cast allowed xs:gMonthDay = N
 [174] xs:gYear cast allowed xs:gDay = N
 [175] xs:gYear cast allowed xs:gMonth = N
 [176] xs:gYear cast allowed xs:boolean = N
 [177] xs:gYear cast allowed xs:base64Binary = N
 [178] xs:gYear cast allowed xs:hexBinary = N
 [179] xs:gYear cast allowed xs:anyURI = N
 [180] xs:gYear cast allowed xs:QN = N
 [181] xs:gYear cast allowed xs:NOTATION = N

[182] xs:gMonthDay cast allowed xs:string = Y
 [183] xs:gMonthDay cast allowed xs:float = N
 [184] xs:gMonthDay cast allowed xs:double = N
 [185] xs:gMonthDay cast allowed xs:decimal = N
 [186] xs:gMonthDay cast allowed xs:duration = N

[187] xs:gMonthDay cast allowed xs:dateTime = N
 [188] xs:gMonthDay cast allowed xs:time = N
 [189] xs:gMonthDay cast allowed xs:date = N
 [190] xs:gMonthDay cast allowed xs:gYearMonth = N
 [191] xs:gMonthDay cast allowed xs:gYear = N
 [192] xs:gMonthDay cast allowed xs:gDay = N
 [193] xs:gMonthDay cast allowed xs:gMonth = N
 [194] xs:gMonthDay cast allowed xs:boolean = N
 [195] xs:gMonthDay cast allowed xs:base64Binary = N
 [196] xs:gMonthDay cast allowed xs:hexBinary = N
 [197] xs:gMonthDay cast allowed xs:anyURI = N
 [198] xs:gMonthDay cast allowed xs:QN = N
 [199] xs:gMonthDay cast allowed xs:NOTATION = N

[191] xs:gDay cast allowed xs:string = Y
 [192] xs:gDay cast allowed xs:float = N
 [193] xs:gDay cast allowed xs:double = N
 [194] xs:gDay cast allowed xs:decimal = N
 [195] xs:gDay cast allowed xs:duration = N
 [196] xs:gDay cast allowed xs:dateTime = N
 [197] xs:gDay cast allowed xs:time = N
 [198] xs:gDay cast allowed xs:date = N
 [199] xs:gDay cast allowed xs:gYearMonth = N
 [200] xs:gDay cast allowed xs:gYear = N
 [201] xs:gDay cast allowed xs:gMonthDay = N
 [202] xs:gDay cast allowed xs:gMonth = N
 [203] xs:gDay cast allowed xs:boolean = N
 [204] xs:gDay cast allowed xs:base64Binary = N
 [205] xs:gDay cast allowed xs:hexBinary = N
 [206] xs:gDay cast allowed xs:anyURI = N
 [207] xs:gDay cast allowed xs:QN = N
 [208] xs:gDay cast allowed xs:NOTATION = N

[209]	xs:gMonth	cast allowed xs:string	= Y
[210]	xs:gMonth	cast allowed xs:float	= N
[211]	xs:gMonth	cast allowed xs:double	= N
[212]	xs:gMonth	cast allowed xs:decimal	= N
[213]	xs:gMonth	cast allowed xs:duration	= N
[214]	xs:gMonth	cast allowed xs:dateTime	= N
[215]	xs:gMonth	cast allowed xs:time	= N
[216]	xs:gMonth	cast allowed xs:date	= N
[217]	xs:gMonth	cast allowed xs:gYearMonth	= N
[218]	xs:gMonth	cast allowed xs:gYear	= N
[219]	xs:gMonth	cast allowed xs:gMonthDay	= N
[220]	xs:gMonth	cast allowed xs:gDay	= N
[221]	xs:gMonth	cast allowed xs:boolean	= N
[222]	xs:gMonth	cast allowed xs:base64Binary	= N
[223]	xs:gMonth	cast allowed xs:hexBinary	= N
[224]	xs:gMonth	cast allowed xs:anyURI	= N
[225]	xs:gMonth	cast allowed xs:QN	= N
[226]	xs:gMonth	cast allowed xs:NOTATION	= N
[227]	xs:boolean	cast allowed xs:string	= Y
[228]	xs:boolean	cast allowed xs:float	= Y
[229]	xs:boolean	cast allowed xs:double	= Y
[230]	xs:boolean	cast allowed xs:decimal	= Y
[231]	xs:boolean	cast allowed xs:duration	= N
[232]	xs:boolean	cast allowed xs:dateTime	= N
[233]	xs:boolean	cast allowed xs:time	= N
[234]	xs:boolean	cast allowed xs:date	= N
[235]	xs:boolean	cast allowed xs:gYearMonth	= N
[236]	xs:boolean	cast allowed xs:gYear	= N
[237]	xs:boolean	cast allowed xs:MonthDay	= N
[238]	xs:boolean	cast allowed xs:gDay	= N

[239]	xs:boolean cast allowed xs:gMonth	= N
[240]	xs:boolean cast allowed xs:base64Binary	= N
[241]	xs:boolean cast allowed xs:hexBinary	= N
[242]	xs:boolean cast allowed xs:anyURI	= N
[243]	xs:boolean cast allowed xs:QN	= N
[244]	xs:boolean cast allowed xs:NOTATION	= N
[245]	xs:base64Binary cast allowed xs:string	= Y
[246]	xs:base64Binary cast allowed xs:float	= N
[247]	xs:base64Binary cast allowed xs:double	= N
[248]	xs:base64Binary cast allowed xs:decimal	= N
[249]	xs:base64Binary cast allowed xs:duration	= N
[250]	xs:base64Binary cast allowed xs:dateTime	= N
[251]	xs:base64Binary cast allowed xs:time	= N
[252]	xs:base64Binary cast allowed xs:date	= N
[253]	xs:base64Binary cast allowed xs:gYearMonth	= N
[254]	xs:base64Binary cast allowed xs:gYear	= N
[255]	xs:base64Binary cast allowed xs:MonthDay	= N
[256]	xs:base64Binary cast allowed xs:gDay	= N
[257]	xs:base64Binary cast allowed xs:gMonth	= N
[258]	xs:base64Binary cast allowed xs:boolean	= M
[259]	xs:base64Binary cast allowed xs:hexBinary	= N
[260]	xs:base64Binary cast allowed xs:anyURI	= N
[261]	xs:base64Binary cast allowed xs:QN	= N
[262]	xs:base64Binary cast allowed xs:NOTATION	= N
[263]	xs:hexBinary cast allowed xs:string	= Y
[264]	xs:hexBinary cast allowed xs:float	= M
[265]	xs:hexBinary cast allowed xs:double	= M
[266]	xs:hexBinary cast allowed xs:decimal	= M
[267]	xs:hexBinary cast allowed xs:duration	= N
[268]	xs:hexBinary cast allowed xs:dateTime	= N

[269]	xs:hexBinary cast allowed xs:time	= N
[270]	xs:hexBinary cast allowed xs:date	= N
[271]	xs:hexBinary cast allowed xs:gYearMonth	= N
[272]	xs:hexBinary cast allowed xs:gYear	= N
[273]	xs:hexBinary cast allowed xs:MonthDay	= N
[274]	xs:hexBinary cast allowed xs:gDay	= N
[275]	xs:hexBinary cast allowed xs:gMonth	= N
[276]	xs:hexBinary cast allowed xs:boolean	= M
[277]	xs:hexBinary cast allowed xs:base64Binary	= N
[278]	xs:hexBinary cast allowed xs:anyURI	= N
[279]	xs:hexBinary cast allowed xs:QN	= N
[280]	xs:hexBinary cast allowed xs:NOTATION	= N
[281]	xs:anyURI cast allowed xs:string	= Y
[282]	xs:anyURI cast allowed xs:float	= N
[283]	xs:anyURI cast allowed xs:double	= N
[284]	xs:anyURI cast allowed xs:decimal	= N
[285]	xs:anyURI cast allowed xs:duration	= N
[286]	xs:anyURI cast allowed xs:dateTime	= N
[287]	xs:anyURI cast allowed xs:time	= N
[288]	xs:anyURI cast allowed xs:date	= N
[289]	xs:anyURI cast allowed xs:gYearMonth	= N
[290]	xs:anyURI cast allowed xs:gYear	= N
[291]	xs:anyURI cast allowed xs:MonthDay	= N
[292]	xs:anyURI cast allowed xs:gDay	= N
[293]	xs:anyURI cast allowed xs:gMonth	= N
[294]	xs:anyURI cast allowed xs:boolean	= N
[295]	xs:anyURI cast allowed xs:base64Binary	= N
[296]	xs:anyURI cast allowed xs:hexBinary	= N
[297]	xs:anyURI cast allowed xs:QN	= N
[298]	xs:anyURI cast allowed xs:NOTATION	= N

[299]	xs:QN cast allowed xs:string	= N
[300]	xs:QN cast allowed xs:float	= N
[301]	xs:QN cast allowed xs:double	= N
[302]	xs:QN cast allowed xs:decimal	= N
[303]	xs:QN cast allowed xs:duration	= N
[304]	xs:QN cast allowed xs:dateTime	= N
[305]	xs:QN cast allowed xs:time	= N
[306]	xs:QN cast allowed xs:date	= N
[307]	xs:QN cast allowed xs:gYearMonth	= N
[308]	xs:QN cast allowed xs:gYear	= N
[309]	xs:QN cast allowed xs:MonthDay	= N
[310]	xs:QN cast allowed xs:gDay	= N
[311]	xs:QN cast allowed xs:gMonth	= N
[312]	xs:QN cast allowed xs:boolean	= N
[313]	xs:QN cast allowed xs:base64Binary	= N
[314]	xs:QN cast allowed xs:hexBinary	= N
[315]	xs:QN cast allowed xs:anyURI	= N
[316]	xs:QN cast allowed xs:NOTATION	= N
[317]	xs:NOTATION cast allowed xs:string	= Y
[318]	xs:NOTATION cast allowed xs:float	= N
[319]	xs:NOTATION cast allowed xs:double	= N
[320]	xs:NOTATION cast allowed xs:decimal	= N
[321]	xs:NOTATION cast allowed xs:duration	= N
[322]	xs:NOTATION cast allowed xs:dateTime	= N
[323]	xs:NOTATION cast allowed xs:time	= N
[324]	xs:NOTATION cast allowed xs:date	= N
[325]	xs:NOTATION cast allowed xs:gYearMonth	= N
[326]	xs:NOTATION cast allowed xs:gYear	= N
[327]	xs:NOTATION cast allowed xs:MonthDay	= N
[328]	xs:NOTATION cast allowed xs:gDay	= N
[329]	xs:NOTATION cast allowed xs:gMonth	= N

```

[330] xs:NOTATION cast allowed xs:boolean      = N
[331] xs:NOTATION cast allowed xs:base64Binary = N
[332] xs:NOTATION cast allowed xs:hexBinary    = N
[333] xs:NOTATION cast allowed xs:anyURI       = N
[334] xs:NOTATION cast allowed xs:QN          = N

```

A.13 AuxiliaryJudgments.sdf e AuxiliaryJudgments.asf

```

module AuxiliaryJudgments
imports AuxiliaryRules
imports DynEnvironment
exports

sorts TList BuiltInAttributes ValueFilter

context-free syntax

("(" TypeName "derives" "from" TypeName "," StatEnvs ")") -> Boolean
("(" ElementName "substitutes" "for" ElementName "," StatEnvs ")") -> Boolean
("(" Type "extended" "by" Type "is" Type "," StatEnvs ")") -> Boolean
("(" Type "mixes" "to" Type "," StatEnvs ")") -> Boolean
("(" Mixed? Type "adjusts" "to" Type "," StatEnvs ")") -> Boolean
("(" TypeSpecifier "resolves" "to" TypeName "{" Type "}"
                                     "," StatEnvs ")") -> Boolean
("(" TypeSpecifier "derives" "from" TypeSpecifier "," StatEnvs ")") -> Boolean
("(" ElementName "lookup" ElementType "yields" Nillable? TypeSpecifier
                                     "," StatEnvs ")") -> Boolean
("(" AttributeName "lookup" AttributeType "yields" TypeSpecifier
                                     "," StatEnvs ")") -> Boolean
("(" Value "interleave" Value "yields" Value "," StatEnvs ")") -> Boolean
("(" Value "nil-matches" Nillable? Type "," StatEnvs ")") -> Boolean
("(" Value "matches" TList "," StatEnvs ")") -> Boolean
"eval" "(" Value "on" Filter "," DynEnvs ")" -> Value
"eval" "(" Value "filter" "@" QName "," (DynEnvs | StatEnvs) ")" -> ValueFilter

```

```

SimpleValue                                -> ValueFilter
"(")" "or" SimpleValue                    -> ValueFilter
Type | {Type ","}* | {Type ","}+ | Type?  -> TList
BuiltInAttributes                          -> Type
(("attribute" "xsi" ":" "type") |
 ("attribute" "xsi" ":" "nil") |
 ("attribute" "xsi" ":" "schemaLocation") |
 ("attribute" "xsi" ":" "noNamespaceSchemaLocation"))+ -> BuiltInAttributes

variables
  "BuiltInAttr" -> BuiltInAttributes

equations

[deri-1] (TName derives from TName, statEnvs) = true

[deri-2]
  lookup(TName, type-defn(statEnvs)) =
    define type TName extends BaseTName Mix {TOpc}
    =====
    (TName derives from BaseTName, statEnvs) = true

[deri-3]
  lookup(TName, type-defn(statEnvs)) =
    define type TName restricts BaseTName Mix {TOpc}
    =====
    (TName derives from BaseTName, statEnvs) = true

[deri-4]
  (TName1 derives from TName2, statEnvs) = true,
  (TName2 derives from TName3, statEnvs) = true
  =====
  (TName1 derives from TName3, statEnvs) = true

```

[subs-1] (ElemName substitutes for ElemName, statEnvs) = true

[subs-2]

```
lookup(ElemName, elem-decl(statEnvs)) = define
    element ElemName substitutes for BaseElemName Nil1 TSpec
=====
(ElemName substitutes for BaseElemName, statEnvs) = true
```

[subs-3]

```
(ElemName1 substitutes for ElemName2, statEnvs) = true,
(ElemName2 substitutes for ElemName3, statEnvs) = true
=====
(ElemName1 substitutes for ElemName3, statEnvs) = true
```

[ext]

```
T1 = (AttrAll1, ElemCont1),
T2 = (AttrAll2, ElemCont2)
=====
(T1 extended by T2 is (AttrAll1 & AttrAll2),
    ElemCont1, ElemCont2, statEnvs) = true
```

[mxs]

```
T = (AttrAll, ElemCont)
=====
(T mixes to (AttrAll, (ElemCont & text)), statEnvs) = true
```

[adj-1]

```
(T mixes to T1, statEnvs) = true,
(T1 extended by BuiltInAttr is T2, statEnvs) = true
=====
(mixed T adjusts to T2, statEnvs) = true
```

[adj-2]

(T extended by BuiltInAttr is T1, statEnvs) = true

=====

(T adjusts to T1, statEnvs) = true

[res-1]

(Deriv Mix {} resolves to TName {T}, statEnvs) = true

=====

(Deriv Mix {} resolves to TName {T}, statEnvs) = true

[res-2]

lookup(TName, type-defn(statEnvs)) = define type TName TDeriv,

(TDeriv resolves to BaseTName {T}, statEnvs) = true

=====

(of type TName resolves to TName {T}, statEnvs) = true

[res-3]

(Mix T adjusts to AdjT, statEnvs) = true

=====

(restricts TName Mix {T} resolves to TName {AdjT}, statEnvs) = true

[res-4]

lookup(TName, type-defn(statEnvs)) =

define type TName Deriv BaseMix {BaseT},

(BaseT extended by T is ExtT, statEnvs) = true,

(Mix ExtT adjusts to AdjT, statEnvs) = true

=====

(extends TName Mix {T} resolves to TName {AdjT}, statEnvs) = true

[der]

(TSpec resolves to TName {T}, statEnvs) = true,

```

(TName derives from TName1, statEnvs) = true
=====
(TSpec derives from of type TName1, statEnvs) = true

[look-1]
  (ElemName substitutes for ElemName1, statEnvs) = true,
  lookup(ElemName, elem-decl(statEnvs)) =
    define element ElemName Substitution Nill TSpec
    =====
    (ElemName lookup element ElemName1 yields Nill TSpec, statEnvs) = true

[look-2] (ElemName lookup element ElemName Nill TSpec
          yields Nill TSpec, statEnvs) = true

[look-3] (ElemName lookup element TSpec yields TSpec, statEnvs) = true

[look-4] (ElemName lookup element yields of type xs:anyType, statEnvs) = true

[look-a]
  lookup(AttrName, attr-decl(statEnvs)) = define attribute AttrName TSpec
  =====
  (AttrName lookup attribute AttrName yields TSpec, statEnvs) = true

[look-b] (AttrName lookup attribute AttrName TSpec yields TSpec, statEnvs) = true

[look-c] (AttrName lookup attribute TSpec yields TSpec, statEnvs) = true

[look-d] (AttrName lookup attribute yields of type xs:anySimpleType, statEnvs) = true

[intr-1] (() interleave () yields (), statEnvs) = true

[intr-2]

```

```

(Val1 interleave Val2 yields Val, statEnvs) = true
=====
((I, Val1) interleave Val2 yields (I, Val), statEnvs) = true

[intr-3]
(Val1 interleave Val2 yields Val, statEnvs) = true
=====
(Val1 interleave (I, Val2) yields (I, Val), statEnvs) = true

[filt-1]
eval(Val on attribute::, dynEnvs) = Val1,
eval(Val1 on (attribute, QN), dynEnvs) = ()
=====
eval(Val filter @QN, dynEnvs) = ()

[filt-2]
eval(Val on attribute::, dynEnvs) = Val1,
eval(Val1 on (attribute, QN), dynEnvs) = Val2,
Val2 = attribute QN {SimpleVal}
=====
eval(Val filter @QN, dynEnvs) = SimpleVal

[nil-1]
eval(Val filter @xsi:nil, statEnvs) = (),
(Val matches T, statEnvs) = true
=====
(Val nil-matches T, statEnvs) = true

[nil-2]
eval(Val filter @xsi:nil, statEnvs) = () or false,
(Val matches T, statEnvs) = true
=====

```

```

(Val nil-matches nillable T, statEnvs) = true

[nil-3]
  eval(Val filter @xsi:nil, statEnvs) = true,
  (Val matches AttrAll, statEnvs) = true
  =====
  (Val nil-matches nillable (AttrAll, ElemCont), statEnvs) = true

[mtch-1] (() matches (), statEnvs) = true

[mtch-2]
  (Val1 matches T1, statEnvs) = true,
  (Val2 matches T2, statEnvs) = true
  =====
  ((Val1, Val2) matches (T1, T2), statEnvs) = true

[mtch-3]
  (Val matches T1, statEnvs) = true
  =====
  (Val matches (T1 | T2), statEnvs) = true

[mtch-4]
  (Val matches T2, statEnvs) = true
  =====
  (Val matches (T1 | T2), statEnvs) = true

[mtch-5]
  (Val1 matches T1, statEnvs) = true,
  (Val2 matches T2, statEnvs) = true,
  (Val1 interleave Val2 yields Val, statEnvs) = true
  =====
  (Val1 matches (T1 & T2), statEnvs) = true

```

```

[mtch-6]
    (Val matches (T | ()), statEnvs) = true
    =====
    (Val matches TOpc, statEnvs) = true

[mtch-7] (( ) matches Ts, statEnvs) = true

[mtch-8]
    (Val1 matches T, statEnvs) = true,
    (Val2 matches Ts, statEnvs) = true
    =====
    ((Val1, Val2) matches Ts, statEnvs) = true

[mtch-9]
    (Val1 matches T, statEnvs) = true,
    (Val2 matches Ts, statEnvs) = true
    =====
    ((Val1, Val2) matches TPlus, statEnvs) = true

[mtch-10]
    (ElemName lookup ElemT yields Nill TSpec, statEnvs) = true,
    (TSpec resolves to TName1 {T}, statEnvs) = true,
    (TName derives from TName1, statEnvs) = true,
    (Val nil-matches Nill T, statEnvs) = true
    =====
    (element ElemName of type TName {Val} matches ElemT, statEnvs) = true

[mtch-11]
    (AttrName lookup AttrT yields TSpec, statEnvs) = true,
    (TSpec resolves to TName1 {T}, statEnvs) = true,
    (TName derives from TName1, statEnvs) = true,

```

```

(Val nil-matches Nill T, statEnvs) = true
=====
(attribute AttrName of type TName {SimpleVal} matches AttrT, statEnvs) = true

[mtch-12]
(Val matches T, statEnvs) = true
=====
(document {Val} matches document {T}, statEnvs) = true

[mtch-13] (text {StringLit} matches text, statEnvs) = true

[mtch-14]
(AtomicT derives from AtomicT1, statEnvs) = true
=====
(AtomicVal of type AtomicT matches AtomicT1, statEnvs) = true

[mtch-15] (AtomicVal of type xs:AnySimpleType matches untyped, statEnvs) = true

```

A.14 AuxiliaryRules.sdf e AuxiliaryRules.asf

```

module AuxiliaryRules
imports StatEnvironment
exports
  sorts PrincipalNodeKind Axis Filter LocalTypeEnv
      ListOfPairsLocal PairLocal AlternativeType

context-free syntax
  "element" | "attribute" | "namespace"          -> PrincipalNodeKind
  ForwardAxis | ReverseAxis                      -> Axis
  Axis "principal" PrincipalNodeKind             -> Boolean
  Axis | ("("PrincipalNodeKind "," NodeTest")") -> Filter
  ListOfPairsLocal                              -> LocalTypeEnv

```

"[" PairLocal* "]"	-> ListOfPairsLocal
"(" QName "-->" Type ")"	-> PairLocal
"lookup-local" "(" QName "," LocalTypeEnv ")"	-> Type
"insert-local" PairLocal "of" LocalTypeEnv	-> LocalTypeEnv
"extract-type-local" "(" LocalTypeEnv ")"	-> Type
"extract-types-local" "(" LocalTypeEnv ")"	-> ListType
{Type " " }*	-> AlternativeType
"[" AlternativeType "]"	-> AlternativeType
"modify" ListType "to" AlternativeType	-> AlternativeType

variables

"F"	-> Filter
"Ax"	-> Axis
"PNK"	-> PrincipalNodeKind
"localTypeEnv" [0-9\']*	-> LocalTypeEnv
"PLs"	-> PairLocal*
"PL"	-> PairLocal
"AltT"	-> AlternativeType

equations

[locenv-1] lookup-local(QN, []) = error

[locenv-2] lookup-local(QN, [(QN --> T) PLs]) = T

[locenv-3]

```

QN != QN1,
lookup-local(QN, [PLs]) = T2
=====
lookup-local(QN, [(QN1 --> T1) PLs]) = T2

```

[locenv-4]

```

insert-local PL of [PLs] = [PL PLs]

```

```

[locenv-5]
    extract-type-local([(QN --> T)]) = T

[locenv-6]
    extract-types-local([]) = []

[locenv-7]
    extract-types-local([(QN --> T)]) = [T]

[locenv-8]
    extract-type-local([PL]) = T,
    extract-types-local([PLs]) = [Ts]
    =====
    extract-types-local([PL PLs]) = [T, Ts]

[alt-1]
    modify [] to AltT = AltT

[alt-2]
    modify [T] to [] = [T]

[alt-3]
    modify [T1, T2] to [] = [T1 | T2]

[alt-4]
    modify [Ts] to AltT = [TsOr]
    =====
    modify [T, Ts] to AltT = [T | TsOr]

```

A.15 SeqTypesNormalization.sdf e SeqTypesNormalization.asf

```
module SeqTypesNormalization
imports StatEnvironment
exports
  sorts ListSeqType ListTypeOr

context-free syntax
  ("normSeqT" "(" SequenceType ")") -> Type {prefer}
  ("normSeqT-list" "(" ListSeqType ")") -> ListTypeOr {prefer}
  "[" {SequenceType ","}* "]" -> ListSeqType
  "[" {Type "|"}* "]" -> ListTypeOr

equations

[seqt-1]
  normSeqT(IT1) = T
  =====
  normSeqT(IT1 OI1) = T OI1

[seqt-2] normSeqT(empty) = ()

[seqt-3] normSeqT(element SchemaT) = element SchemaT

[seqt-4] normSeqT(attribute SchemaT) = attribute SchemaT

[seqt-5] normSeqT(element QN) = element QN

[seqt-6] normSeqT(attribute QN) = attribute QN

[seqt-7] normSeqT(text) = (text)
```

[seqt-8] normSeqT(document) = (document)

[seqt-9] normSeqT(untyped) = (untyped)

[seqt-10] normSeqT(AtomicT) = (AtomicT)

[seqt-11] normSeqT(node) = (element)

[seqt-12] normSeqT(node) = (attribute)

[seqt-13] normSeqT(node) = (text)

[seqt-14] normSeqT(node) = (document)

[seqt-15] normSeqT(item) = (element)

[seqt-16] normSeqT(item) = (attribute)

[seqt-17] normSeqT(item) = (text)

[seqt-18] normSeqT(item) = (document)

[seqt-19] normSeqT(item) = xs:string

[seqt-20] normSeqT(item) = xs:decimal

[seqt-21] normSeqT(item) = xs:integer

[seqt-22] normSeqT(item) = xs:double

[seqt-23] normSeqT(item) = xs:boolean

[seqt-24] normSeqT(item) = xs:float

[seqt-25] normSeqT(item) = xs:duration

[seqt-26] normSeqT(item) = xs:dateTime

[seqt-27] normSeqT(item) = xs:time

[seqt-28] normSeqT(item) = xs:date

[seqt-29] normSeqT(item) = xs:gYearMonth

[seqt-30] normSeqT(item) = xs:gYear

[seqt-31] normSeqT(item) = xs:gMonthDay

[seqt-32] normSeqT(item) = xs:gDay

[seqt-33] normSeqT(item) = xs:gMonth

[seqt-34] normSeqT(item) = xs:hexBinary

```

[seqt-35] normSeqT(item) = xs:base64Binary
[seqt-36] normSeqT(item) = xs:anyURI
[seqt-37] normSeqT(item) = xs:NOTATION

[seqt-38] normSeqT(atomic value) = xs:string
[seqt-39] normSeqT(atomic value) = xs:decimal
[seqt-40] normSeqT(atomic value) = xs:integer
[seqt-41] normSeqT(atomic value) = xs:double
[seqt-42] normSeqT(atomic value) = xs:boolean
[seqt-43] normSeqT(atomic value) = xs:float
[seqt-44] normSeqT(atomic value) = xs:duration
[seqt-45] normSeqT(atomic value) = xs:dateTime
[seqt-46] normSeqT(atomic value) = xs:time
[seqt-47] normSeqT(atomic value) = xs:date
[seqt-48] normSeqT(atomic value) = xs:gYearMonth
[seqt-49] normSeqT(atomic value) = xs:gYear
[seqt-50] normSeqT(atomic value) = xs:gMonthDay
[seqt-51] normSeqT(atomic value) = xs:gDay
[seqt-52] normSeqT(atomic value) = xs:gMonth
[seqt-53] normSeqT(atomic value) = xs:hexBinary
[seqt-54] normSeqT(atomic value) = xs:base64Binary
[seqt-55] normSeqT(atomic value) = xs:anyURI
[seqt-56] normSeqT(atomic value) = xs:NOTATION

[seqtl-1] normSeqT-list([]) = []

[seqtl-1]
    normSeqT(SeqT) = T,
    normSeqT-list([SeqTs]) = [TsOr]
    =====
    normSeqT-list([SeqT, SeqTs]) = [T | TsOr]

```

A.16 XQueryStaticTyping.sdf e XQueryStaticTyping.asf

```

module XQueryStaticTyping
imports PrimeTypes SeqTypesNormalization FunctionsOnQNames CastingTable
exports
  sorts Operation ArithmeticExpr ListExpr

context-free syntax
  "typecheck" "(" QueryBody "," StatEnvs ")" -> Type {avoid}
  "typecheck" "(" ExprSequence "," StatEnvs ")" -> ListType
  "typecheck" "(" ParenthesizedExpr "," StatEnvs ")" -> ListType
  "typecheck" "(" ArithmeticExpr "," StatEnvs ")" -> Type {prefer}
  "typecheck" "(" Filter "," StatEnvs ")" -> Type
  "typecheck" "(" Axis NodeTest "," StatEnvs ")" -> Type {prefer}
  "typecheck" "(" Filter "on" Type "," StatEnvs ")" -> Type
  "typecheck" "(" Filter "on" Type "," StatEnvs
    "," LocalTypeEnv ")" -> AlternativeType
  "typecheck-list" "(" ListExpr "," StatEnvs ")" -> ListType
  "[" {Expr ","}* "]" -> ListExpr
  Operation "(" Expr "," Expr ")" -> ArithmeticExpr {prefer}
  Operation "(" Expr ")" -> ArithmeticExpr {prefer}
  "op"::"numeric-add" -> Operation
  "op"::"numeric-subtract" -> Operation
  "op"::"numeric-unary-minus" -> Operation
  PICO-INT -> ArithmeticExpr

variables
  "op"::"operation" -> Operation

equations

```

%%Primary Expressions-----

```
[lite-1] typecheck(IntegerLit, statEnvs) = xs:integer
```

```
[lite-2] typecheck(DecimalLit, statEnvs) = xs:decimal
```

```
[lite-3] typecheck(DoubleLit, statEnvs) = xs:double
```

```
[lite-4] typecheck(StringLit, statEnvs) = xs:string
```

```
[var]
```

```
    lookup($ Var, var-type(statEnvs)) = T
    =====
    typecheck($ Var, statEnvs) = T
```

```
[func]
```

```
    FuncCall = QN(E1, Es),
    lookup((QN), func-type(statEnvs)) =
        define function QN(T1, Ts1) returns T,
    typecheck-list([E1, Es], statEnvs) = [T2, Ts2],
    subtype-list([T2, Ts2], [T1, Ts1], statEnvs) = true
    =====
    typecheck(FuncCall, statEnvs) = T
```

```
[default-func] typecheck(FuncCall, statEnvs) = static error
```

```
%%Path Expressions-----
```

```
[path-1] attribute:: principal attribute = true
```

```
[path-2] namespace:: principal namespace = true
```

```
[path-3]
```

```
    Ax != attribute::,
```

```

Ax != namespace::
=====

Ax principal element = true

[default-path] Ax principal PNK = false

[axnode-1a]
    %nesta regra PNK (3a linha) foi substituido por element%
    lookup($fs:dot, var-type(statEnvs)) = T1,
    typecheck(Ax on T1, statEnvs) = T2,
    Ax principal element = true,
    typecheck((element, NTest) on T2, statEnvs) = T3
    =====
    typecheck(Ax NTest, statEnvs) = T3

[axnode-1b]
    %nesta regra PNK (3a linha) foi substituido por attribute%
    lookup($fs:dot, var-type(statEnvs)) = T1,
    typecheck(Ax on T1, statEnvs) = T2,
    Ax principal attribute = true,
    typecheck((attribute, NTest) on T2, statEnvs) = T3
    =====
    typecheck(Ax NTest, statEnvs) = T3

[axnode-1c]
    %nesta regra PNK (3a linha) foi substituido por namespace%
    lookup($fs:dot, var-type(statEnvs)) = T1,
    typecheck(Ax on T1, statEnvs) = T2,
    Ax principal namespace = true,
    typecheck((namespace, NTest) on T2, statEnvs) = T3
    =====
    typecheck(Ax NTest, statEnvs) = T3

```

[axnode-2] typecheck(F on (), statEnvs) = ()

[axnode-3] typecheck(F on none, statEnvs) = none

[axnode-4]

typecheck(F on T1, statEnvs) = T3,

typecheck(F on T2, statEnvs) = T4

=====

typecheck(F on T1 , T2, statEnvs) = (T3 , T4)

[axnode-5]

typecheck(F on T1, statEnvs) = T3,

typecheck(F on T2, statEnvs) = T4

=====

typecheck(F on T1 | T2, statEnvs) = T3 | T4

[axnode-6]

typecheck(F on T1, statEnvs) = T3,

typecheck(F on T2, statEnvs) = T4

=====

typecheck(F on T1 & T2, statEnvs) = T3 & T4

[axes-1] typecheck(self:: on NodeT, statEnvs) = NodeT

[axes-2] typecheck(child:: on element QN{AttrT, ChildT}, statEnvs) = ChildT

[axes-3] typecheck(attribute:: on element QN{AttrT, ChildT}, statEnvs) = AttrT

[axes-4] typecheck(parent:: on element, statEnvs) = (element | document) | none

[axes-5] typecheck(namespace:: on NodeT, statEnvs) = ()

[axes-6]

```
localTypeEnv = [],  
typecheck(descendant:: on T, statEnvs, localTypeEnv) = T1  
=====  
typecheck(descendant:: on T, statEnvs) = T1
```

[axes-7]

```
lookup-local(QN, localTypeEnv) = T,  
T != error,  
extract-types-local(localTypeEnv) = [Ts],  
modify [Ts] to [] = AltT  
=====  
typecheck(descendant:: on element QN, statEnvs, localTypeEnv) = AltT
```

[axes-8]

```
lookup-local(QN, localTypeEnv) = error,  
lookup((QN), elem-decl(statEnvs)) = Def,  
Def = define element QN TSpec,  
(TSpec resolves to TName {T}, statEnvs) = true,  
insert-local (QN --> T) of localTypeEnv = localTypeEnv1,  
typecheck(descendant:: on T, statEnvs, localTypeEnv1) = AltT  
=====  
typecheck(descendant:: on element QN, statEnvs, localTypeEnv) = AltT
```

[axes-9a]

```
typecheck(child:: on NodeT, statEnvs) = T1,  
typecheck(descendant:: on T1, statEnvs) = T2  
=====  
typecheck(descendant:: on NodeT, statEnvs) = (T1 , T2)
```

[axes-9b]

```

typecheck(self:: on NodeT, statEnvs) = T1,
typecheck(descendant:: on T1, statEnvs) = T2
=====
typecheck(descendant-or-self:: on NodeT, statEnvs) = (T1 , T2)

[axes-10a] typecheck(ancestor:: on NodeT, statEnvs) = element | ()

[axes-10b]
typecheck(self:: on NodeT, statEnvs) = T1,
typecheck(ancestor:: on T1, statEnvs) = T2
=====
typecheck(ancestor-or-self:: on NodeT, statEnvs) = (T1 , T2)

[axes-11] typecheck(following-sibling:: on NodeT, statEnvs) = ()

[axes-12] typecheck(following:: on NodeT, statEnvs) = ()

[axes-13] typecheck(preceding-sibling:: on NodeT, statEnvs) = ()

[axes-14] typecheck(preceding:: on NodeT, statEnvs) = ()

[node-1]
xf:get-namespace-uri(QN1) = NCN,
lookup(Pre2, namespace(statEnvs)) = NCN,
xf:get-local-name(QN1) = NCN1,
=====
typecheck((element, Pre2:NCN1) on element QN1 {T}, statEnvs) =
                                                    (element QN1 {T})

[node-2]
NCN1 = NCN2,
=====

```

```

typecheck((element, Pre2:NCN2) on element *:NCN1 {T}, statEnvs) =
                                                    (element Pre2:NCN2 {T})

```

[node-3]

```

lookup(NCN1, namespace(statEnvs)) = lookup(Pre2, namespace(statEnvs))
=====
typecheck((element, Pre2:Loc2) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:Loc2 {T})

```

[node-4]

```

typecheck((element, QN2) on element {T}, statEnvs) = (element QN2 {T})

```

[node-5]

```

xf:get-local-name(QN1) = NCN2
=====
typecheck((element, *:NCN2) on element QN1 {T}, statEnvs) = (element QN1 {T})

```

[node-6]

```

NCN1 = NCN2
=====
typecheck((element, *:NCN2) on element *:NCN1 {T}, statEnvs) =
                                                    (element *:NCN2 {T})

```

```

[node-7] typecheck((element, *:NCN2) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:NCN2 {T})

```

```

[node-8] typecheck((element, *:NCN2) on element {T}, statEnvs) =
                                                    (element *:NCN2 {T})

```

[node-9]

```

xf:get-namespace-uri(QN1) = lookup(NCN2, namespace(statEnvs))
=====

```

```

typecheck((element, NCN2:*) on element QN1 {T}, statEnvs) = (element QN1 {T})

[node-10] typecheck((element, NCN2:*) on element *:NCN1 {T}, statEnvs) =
                                                    (element NCN2:NCN1 {T})

[node-11]
lookup(NCN1, namespace(statEnvs)) = lookup(NCN2, namespace(statEnvs))
=====
typecheck((element, NCN2:*) on element NCN1:* {T}, statEnvs) =
                                                    (element NCN1:* {T})

[node-12] typecheck((element, NCN2:*) on element {T}, statEnvs) =
                                                    (element NCN2:* {T})

[node-13] typecheck((element, *) on element Pre:Loc {T}, statEnvs) =
                                                    (element Pre:Loc {T})

[node-14]
xf:get-namespace-uri(QN1) = lookup(Pre2, namespace(statEnvs)),
xf:get-local-name(QN1) = NCN1
=====
typecheck((attribute, Pre2:NCN1) on attribute QN1 {T}, statEnvs) =
                                                    (attribute QN1 {T})

[node-15]
NCN1 = NCN2
=====
typecheck((attribute, Pre2:NCN2) on attribute *:NCN1 {T}, statEnvs) =
                                                    (attribute Pre2:NCN2 {T})

[node-16]
lookup(Pre1, namespace(statEnvs)) = lookup(Pre2, namespace(statEnvs))
=====

```

```

typecheck((attribute, Pre2:Loc2) on attribute Pre1:* {T}, statEnvs) =
                                                    (attribute Pre1:Loc2 {T})

[node-17] typecheck((attribute, QN2) on attribute {T}, statEnvs) =
                                                    (attribute QN2 {T})

[node-18]
xf:get-local-name(QN1) = NCN1
=====
typecheck((attribute, Pre:NCN1) on attribute QN1 {T}, statEnvs) =
                                                    (attribute QN1 {T})

[node-19]
NCN1 = NCN2
=====
typecheck((attribute, *:NCN2) on attribute *:NCN1 {T}, statEnvs) =
                                                    (attribute *:NCN2 {T})

[node-20] typecheck((attribute, *:NCN2) on attribute NCN1:* {T}, statEnvs) =
                                                    (attribute NCN1:NCN2 {T})

[node-21] typecheck((attribute, *:NCN2) on attribute {T}, statEnvs) =
                                                    (attribute *:NCN2 {T})

[node-22]
xf:get-namespace-uri(QN1) = lookup(NCN2, namespace(statEnvs))
=====
typecheck((attribute, NCN2:*) on attribute QN1 {T}, statEnvs) =
                                                    (attribute QN1 {T})

[node-23] typecheck((attribute, NCN2:*) on attribute *:NCN1 {T}, statEnvs) =
                                                    (attribute NCN2:NCN1 {T})

```

[node-24]

```
lookup(NCN1, namespace(statEnvs)) = lookup(NCN2, namespace(statEnvs))
=====
typecheck((attribute, NCN2:*) on attribute NCN1:* {T}, statEnvs) =
                                                    (attribute NCN1:* {T})
```

[node-25] typecheck((attribute, NCN2:*) on attribute {T}, statEnvs) =

(attribute NCN2:* {T})

[node-26] typecheck((attribute, *) on attribute Pre:Loc {T}, statEnvs) =

(attribute Pre:Loc {T})

[node-27] typecheck((PNK, processing-instruction()) on processing-instruction,

statEnvs) = (processing-instruction)

[node-28] typecheck((PNK, processing-instruction(StringLit))

on processing-instruction, statEnvs) = (processing-instruction)

[node-29] typecheck((PNK, comment()) on comment, statEnvs) = (comment)

[node-30] typecheck((PNK, text()) on text, statEnvs) = (text)

[node-31] typecheck((PNK, node()) on NodeT, statEnvs) = NodeT

%%Sequence Expressions-----

[sequ]

```
typecheck-list([E, Es], statEnvs) = [Ts]
```

=====

```
typecheck(E, Es, statEnvs) = [Ts]
```

[paren]

```
typecheck(E, Es, statEnvs) = LT
=====
typecheck((E, Es), statEnvs) = LT
```

[lst-1] typecheck-list([], statEnvs) = [()]

[lst-2]

```
typecheck(E, statEnvs) = T
=====
typecheck-list([E], statEnvs) = [T]
```

[lst-3]

```
typecheck(E, statEnvs) = T,
typecheck-list([Es], statEnvs) = [Ts]
=====
typecheck-list([E, Es], statEnvs) = [T, Ts]
```

%%Arithmetic Expressions-----

[arith-1]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:decimal, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:decimal, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:decimal
```

[arith-2]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:float, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
```

```

subtype(T2, xs:float, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:float

```

[arith-3]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:double, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:double, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:double

```

[arith-4]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:integer, statEnvs) = true,
typecheck(E2, statEnvs) = T2,
subtype(T2, xs:integer, statEnvs) = true
=====
typecheck(op:operation(E1, E2), statEnvs) = xs:integer

```

[arith-5]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:decimal, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:decimal

```

[arith-6]

```

typecheck(E1, statEnvs) = T1,
subtype(T1, xs:float, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:float

```

[arith-7]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:double, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:double
```

[arith-8]

```
typecheck(E1, statEnvs) = T1,
subtype(T1, xs:integer, statEnvs) = true
=====
typecheck(op:operation(E1), statEnvs) = xs:integer
```

%%Constructors-----

[const-1]

```
expand(QN, statEnvs) = qname
=====
typecheck(element QN {ExprSeq}, statEnvs) = element qname {xs:anyType}
```

[const-2]

```
typecheck(E, statEnvs) = xs:NCN
=====
typecheck(element {E} {ExprSeq}, statEnvs) = element {xs:anyType}
```

[const-3]

```
typecheck(E, statEnvs) = xs:NCN
=====
typecheck(attribute {E} {ExprSeq}, statEnvs) = attribute {xs:anySimpleType}
```

[const-4]

```
expand(QN, statEnvs) = qname
```

```

=====
typecheck(attribute QN {ExprSeq}, statEnvs)
                                = attribute qname {xs:anySimpleType}

[default-const] typecheck(Constr, statEnvs) = static error

%%FLWR Expressions-----

[for-1]
    typecheck(E1, statEnvs) = T1,
    insert($ Var1 --> prime(T1)) in VT of statEnvs = statEnvs1,
    typecheck(TExpr, statEnvs1) = T2
    =====
    typecheck(for $ Var1 in E1 return TExpr, statEnvs) = T2.quantifier(T1)

[for-2]
    typecheck(E1, statEnvs) = T1,
    T0 = normSeqT(SeqT),
    subtype(T1, T0, statEnvs) = true,
    insert($ Var1 --> prime(T1)) in VT of statEnvs = statEnvs1,
    typecheck(TExpr, statEnvs1) = T2
    =====
    typecheck(for SeqT $ Var1 in E1 return TExpr, statEnvs) = T2.quantifier(T1)

[default-for] typecheck(For, statEnvs) = static error

[let-1]
    typecheck(E1, statEnvs) = T1,
    insert($ Var1 --> T1) in VT of statEnvs = statEnvs1,
    typecheck(TExpr, statEnvs1) = T2
    =====
    typecheck(let $ Var1 := E1 return TExpr, statEnvs) = T2

```

[let-2]

```
typecheck(E1, statEnvs) = T1,
T0 = normSeqT(SeqT),
subtype(T1, T0, statEnvs) = true,
insert($ Var1 --> T1) in VT of statEnvs = statEnvs1,
typecheck(TExpr, statEnvs1) = T2
=====
typecheck(let SeqT $ Var1 := E1 return TExpr, statEnvs) = T2
```

[default-let] typecheck(Let, statEnvs) = static error

%%Order-Related Expressions-----

[unord]

```
typecheck(FLEExpr, statEnvs) = T1
=====
typecheck((unordered FLEExpr), statEnvs) = prime(T1).quantifier(T1)
```

[default-unord] typecheck(UnorExpr, statEnvs) = static error

%%Conditional Expressions-----

[if]

```
typecheck(E1, statEnvs) = xs:boolean,
typecheck(E2, statEnvs) = T2,
typecheck(ValExpr, statEnvs) = T3
=====
typecheck((if (E1) then E2 else ValExpr), statEnvs) = (T2 | T3)
```

[cast-1a]

```
typecheck(E, statEnvs) = T1,
```

```

quantifier(T1) = ?,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1b]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1c]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = ?,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-1d]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) = *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = (T2 | none)

```

[cast-2a]

```

typecheck(E, statEnvs) = T1,

```

```

quantifier(T1) != ?,
quantifier(T1) != *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = Y
=====
typecheck(cast as SeqT2 (E), statEnvs) = T2

```

[cast-2b]

```

typecheck(E, statEnvs) = T1,
quantifier(T1) != ?,
quantifier(T1) != *,
T2 = normSeqT(SeqT2),
T1 cast allowed T2 = M
=====
typecheck(cast as SeqT2 (E), statEnvs) = T2

```