6.3 Weitere Ausdrücke

Neben Vergleichsausdrücken erlaubt XQuery die Angabe von logischen, quantifizierenden und konditionalen Ausdrücken, die im Rahmen dieses Abschnitts aufgearbeitet werden.

6.3.1 Logische Ausdrücke

In XQuery wird ein logischer Ausdruck entweder aus einem `and`- oder einem `or`-Ausdruck gebildet, wobei das nachfolgende Grammatikfragment die Vorrangregelung zwischen `or` und `and` deutlich macht:

```
OrExpr        ::= AndExpr ( or AndExpr )*
AndExpr       ::= InstanceofExpr ( and InstanceofExpr )*
```

or
and

Der effektive boolesche Wert wird dabei für jeden Operanden ermittelt. In Abhängigkeit vom Auftreten eines Laufzeitfehlers bei dieser Ermittlung ergibt sich gemäß Tabelle 6–2 der Wert des entsprechenden logischen Ausdrucks.

AND		**Operand 1**		
		true	**false**	**error**
Operand 2	**true**	true	false	error
	false	false	false	false / error
	error	error	false / error	error
OR		**Operand 1**		
		true	**false**	**error**
Operand 2	**true**	true	true	true / error
	false	true	false	error
	error	true / error	error	error

Tab. 6–2 *Ergebnistabelle für logische Ausdrücke*

Laufzeitfehler bei der Auswertung von logischen Ausdrücken

Werden beide Operanden zu `true` evaluiert, so ergibt sich sowohl für einen `and`- als auch für einen `or`-Ausdruck der effektive boolesche Wert `true`. Tritt bei der Bestimmung des booleschen Wertes auf einer Seite ein Laufzeitfehler auf, so wird mit zwei Ausnahmen der Fehler an die aufrufende Umgebung weitergereicht: Ist ein Operand eines `and`-Operators bereits zu `false` evaluiert worden, kann implementierungsabhängig der `false`-Wert weitergegeben werden, unabhängig davon, ob bei der Auswertung des zweiten Operanden ein Fehler aufgetreten ist

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8

oder nicht. Der analoge Fall gilt auch für die Verknüpfung mit dem or-Operator, wobei nach der Feststellung eines wahren Wertes das Gesamtergebnis zu true evaluiert werden darf, obwohl die Auswertung des zweiten Operators einen Laufzeitfehler hervorgebracht hat bzw. in einem Laufzeitfehler resultieren würde. Die Auswertung logischer Ausdrücke ist somit nicht deterministisch im Fall einer Existenz von Fehlern. Des Weiteren ist zu beachten, dass die beiden Operatoren als Ergebnis den Wert true bzw. false zurückgeben, falls mindestens einer der beiden Operanden die leere Sequenz als Wert aufweist, wobei der effektive boolesche Wert für eine leere Sequenz false ist. Dieses Verhalten steht dabei im Gegensatz zur Semantik bei arithmetischen Operatoren, die in diesem Fall in einer leeren Sequenz resultieren würden.

Der erste der folgenden logischen Ausdrücke, wie sie üblicherweise in der where-Klausel eines FLWOR-Ausdrucks auftreten können, liefert entweder den Wert true oder einen Laufzeitfehler:

```
(: Ausdruck 1 :) "Kurt" ne "Emil" or 4711 idiv 0 = 13
(: Ausdruck 2 :) "Kurt" eq "Kurt" or 4711 idiv 0 = 13
(: Ausdruck 3 :) "Kurt" eq "Kurt" and 4711 idiv 0 = 13
```

In Analogie dazu ist das Ergebnis des zweiten Ausdrucks ebenfalls nicht eindeutig und kann entweder in true oder einem Laufzeitfehler resultieren. Der dritte Ausdruck hingegen muss gezwungenermaßen in einen Laufzeitfehler münden, da für einen and-Ausdruck der zweite Operand zwingend (ohne Fehler) ausgewertet werden muss, sofern der erste Ausdruck zu true evaluiert worden ist.

Funktionen und Operatoren auf booleschen Werten

fn:false()
fn:true()

Neben den logischen Ausdrücken bietet der Sprachumfang von XQuery eine umfassende Sammlung an Funktionen und Operatoren für die Anwendung auf boolesche Werte. Tabelle 6–3 gibt einen Überblick über die interessanten Funktionen in diesem Kontext. Die beiden Funktionen fn:false() und fn:true() liefern den jeweils durch die Bezeichnung suggerierten konstanten booleschen Wert false bzw. true.

fn:not()

Das Fehlen der Negation, die bei den logischen Ausdrücken nicht direkt realisiert ist, wird durch die Funktion fn:not() mit entsprechender Semantik kompensiert. Die Negation ermittelt mit Hilfe der Funktion fn:boolean() (Abschnitt 7.1.3) den effektiven booleschen Wert des Parameters und liefert true, wenn das Ergebnis false bzw. false, wenn der effektive boolesche Wert true ist.

Zu beachten ist, dass die Funktion `fn:not()` sich auf den Wert des Parameters bezieht, welcher durch einen entsprechenden XQuery-Ausdruck ermittelt werden kann. Handelt es sich dabei um einen Pfadausdruck, so kann sich der Test auf die Existenz eines Elementes, und nicht auf dessen Wert beziehen. Zum Beispiel liefert

```
fn:not(.//Medikament)
```

den Wert `false`, falls das Element existiert — unabhängig von der entsprechenden Belegung. Sollte der boolesche Wert negiert werden, so ist ein expliziter Rückgriff auf den Wert des Elementes oder eine explizite Konvertierung in den Datentyp `xs:boolean` sinnvoll:

```
fn:not(fn:data(.//Medikament))
fn:not(xs:boolean(.//Medikament))
```

Die dritte Klasse an Operatoren auf booleschen Werten, die an dieser Stelle erläutert werden, repräsentieren das Spiegelbild der Vergleichsoperatoren für Wertausdrücke auf booleschen Werten. Die Operation `op:boolean-equal()` korrespondiert dabei zum `eq`-Operator, `op:boolean-less-than()` kann als Pendant zum `lt`- bzw. durch Vertauschung der Operanden zum `ge`-Operator betrachtet werden. Analoges gilt für den dritten Vergleichsoperator auf booleschen Werten: `op:boolean-greater-than()`.

Signatur	Beschreibung
`fn:false()` `as xs:boolean`	liefert den Wert false
`fn:true()` `as xs:boolean`	liefert den Wert true

`fn:not(`
`  $arg as item()*)`
`as xs:boolean`
	liefert den Wert true, wenn der effektive boolesche Wert des Argumentes false ist bzw. invertiert

`op:boolean-[equal|`
`            less-than|`
`            greater-than](`
`  $val1 as xs:boolean,`
`  $val2 as xs:boolean)`
`as xs:boolean`
	äquivalent zu Wertevergleichen:
	· $val1 eq $val2
	· $val1 lt $val2 bzw. $val2 ge $val1
	· $val1 gt $val2 bzw. $val2 le $val1

Tab. 6–3 Übersicht von Funktionen und Operatoren auf booleschen Werten

6.3.2 Konditionale Ausdrücke

Konditionale Ausdrücke eröffnen die Möglichkeit einer Fallunterscheidung innerhalb einer XQuery-Anfrage. Da ein konditionaler Ausdruck eine Spezialform eines »normalen« XQuery-Ausdrucks ist, kann

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8

eine dadurch spezifizierte Fallunterscheidung überall dort auftreten, wo ein Ausdruck erlaubt ist. Wie das entsprechende Grammatikfragment zeigt, besteht ein konditionaler Ausdruck aus einem Testausdruck, dessen effektiver boolescher Wert darüber entscheidet, ob das Ergebnis der `then`- bzw. der `else`-Klausel als Ergebnis zurückgeliefert wird.

if (...) then ... else

```
IfExpr        ::= if ( Expr ) then ExprSingle else ExprSingle
```

Der Testausdruck spielt aus Sicht der Anwendung zwei unterschiedliche Rollen. Zum einen werden Vergleichsausdrücke formuliert, um werteabhängig die Auswertung des `then`- bzw. `else`-Teils zu steuern. Der folgende konditionale Ausdruck berechnet beispielsweise die 10-prozentige Zuzahlung zu Medikamenten in der Apotheke in Abhängigkeit vom Preis mit einer unteren Schranke von € 5 bzw. einer oberen Schranke von € 10:

```
for $m in fn:doc("...")
return
  if ( $m/Preis*0.1 < 5.0 ) then 5.0
                           else if ($m/Preis > 100.0) then 10.0
                                                     else $m/Preis * 0.1
```

Zum anderen werden konditionale Ausdrücke eingesetzt, um das Fehlen von Knoten bzw. Attributen bei einer Auswertung zu kompensieren. Dazu gilt, dass, falls der Testausdruck aus einer Sequenz besteht, implizit auf Existenz eines Elementes geprüft wird, wobei der effektive boolesche Wert einer leeren Sequenz `false` ist. Fehlt beispielsweise in dem laufenden Anwendungsszenario die Preisangabe bei einem Medikament, so wird grundsätzlich zu Gunsten der Krankenkasse die höchste Zuzahlung des Patienten fällig:

Test auf Existenz von Knoten bzw. Attributen

```
if ( $m/Preis ) then ... (: obige Zuzahlungsberechnung :)
                else 10.0
```

Der implizite Test auf die Existenz eines Elementes bzw. eines Attributes kann auch explizit durch Rückgriff auf die Funktion `fn:exists()` beschrieben werden:

```
if ( fn:exists($m/Preis) ) then ... (: obige Zuzahlungsberechnung :)
                           else 10.0
```

Im Unterschied zu konditionalen Ausdrücken in anderen Datenbankanfrage- bzw. Programmiersprachen kann der `else`-Teil nicht wegfallen, sondern muss mit angegeben werden. Falls es aus Sicht der Anwendung jedoch keine Alternative geben soll, kann dies beispiels-

weise durch die Rückgabe eines NaN-Wertes simuliert werden. Folgender konditionaler Ausdruck rechnet den Medikamentenpreis – sofern er denn existiert – in US-Dollar um:

```
if ( $m/Preis ) then $m/Preis * 1.21
                else xs:double("NaN")
```

Bei Zeichenketten ist bei einer entsprechenden Reaktion an Stelle von NaN eine leere Zeichenkette zurückzugeben, um die erzwungene Existenz einer else-Klausel zu kompensieren:

```
if ($p//Adresse/Telefon) then fn:concat("+49", $p//Adresse/Telefon)
                         else ""
```

Anwendung konditionaler Ausdrücke

In der Praxis treten konditionale Ausdrücke meist ausschließlich in where- bzw. return-Klauseln von FLWOR-Ausdrücken auf. Folgendes Beispiel ermittelt alle Patienten, bei denen innerhalb eines Vorgangs Medikamente verabreicht wurden, die Kosten in Höhe von mehr als 1000 verursacht haben. Die Kosten ergeben sich dabei aus dem Preis des Medikamentes und – falls vorhanden – der Anzahl der verabreichten Exemplare.

```
let $p := fn:collection("Patienten")
                         //Patient_stationär[@ID="pat_res_010001"]
for $o in $p//Operation//verbrauchter_Artikel
for $a in fn:doc("Verbrauchsartikel.xml")
                                //Artikel[@ID = $o/@Artikel_id]
let $k := if ($o/@Menge) then $a/Einzelpreis * $o/@Menge
                         else $a/Einzelpreis
where $k > 1000
return
  <Medikamentenverbrauch>
    { $p/Name, $a/Bezeichnung, $k }
  </Medikamentenverbrauch>
```

Der konditionale Ausdruck wird in diesem Fall für jeden Vorgang innerhalb der Operation und jeden dabei verbrauchten Artikel zur Berechnung der Gesamtkosten ausgewertet. In vielen Fällen wird ein konditionaler Ausdruck nur einmalig in der return-Klausel zur Generierung des Ergebnisdokumentes verwendet und direkt eingebettet, so dass sich bei Wegfall der where-Bedingung das obige Beispiel wie folgt beschreiben lässt:

```
let $p := fn:collection("Patienten")
                         //Patient_stationär[@ID="pat_res_010001"]
for $o in $p//Operation//verbrauchter_Artikel
for $a in fn:doc("Verbrauchsartikel.xml")
```

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8

```
                                //Artikel[@ID = $o/@Artikel_id]
return
  <Medikamentenverbrauch>
    { $p/Name, $a/Bezeichnung,
      if ($o/@Menge) then $a/Einzelpreis * $o/@Menge
                     else $a/Einzelpreis
    }
  </Medikamentenverbrauch>
```

Eine Schachtelung wird wiederum deutlich, wenn die Gesamtkosten pro Operation berechnet werden sollen. Dabei müssen die Kosten pro verbrauchten Artikel innerhalb einer Operation aufsummiert werden:

```
let $p := fn:collection("Patienten")
                         //Patient_stationär[@ID="pat_res_010001"]
for $o in $p//Operation
return
  <Behandlungsinformation>
    { $p/Name }
    <Datum>{ xs:date($o/Beginn) }</Datum>
    <Kosten>
      { fn:sum(for $x in $o//verbrauchter_Artikel
               for $a in fn:doc("Verbrauchsartikel.xml")//Artikel
               where $a/@ID = $x/@Artikel_id
               return
                  if ($x/@Menge) then $a/Einzelpreis * $x/@Menge
                                 else $a/Einzelpreis )
      }
    </Kosten>
  </Behandlungsinformation>
```

Behandlung von Laufzeitfehlern

Die Auswertung konditionaler Ausdrücke weist – in Analogie zur Auswertung logischer Ausdrücke – eine Besonderheit bei der Behandlung von Laufzeitfehlern auf. Wird der Testausdruck zu `true` evaluiert, so wird der Alternativzweig nicht ausgewertet und eventuell auftretende Laufzeitfehler nicht beachtet. Gleiches gilt für die Auswertung der `then`-Klausel, falls der Testausdruck zu `false` ausgewertet wird.

6.3.3 Quantifizierende Ausdrücke

XQuery unterstützt explizit die Formulierung quantifizierender Ausdrücke durch entsprechende Sprachkonstrukte. Ein quantifizierender Ausdruck besteht dabei minimal aus einem der beiden Schlüsselwörter `some` oder `every` zur Anzeige einer existenziellen bzw. universellen Quantifizierung, einer nur für den Kontext des quantifizierenden Aus-

drucks gültigen Variablenbindung und einem Ausdruck nach dem Schlüsselwort satisfies, gegen den die Belegungen der Variablen getestet werden.

```
QuantifiedExpr  ::= (( some $ ) | ( every $ ))
                      VarName TypeDeclaration? in ExprSingle
                        (, $ VarName TypeDeclaration? in ExprSingle)*
                      satisfies ExprSingle
```

Existenzielle Quantifizierung

some ... in ... satisfies

Handelt es sich um eine existenzielle Quantifizierung, eingeleitet mit dem Schlüsselwort some, so wird der quantifizierende Ausdruck zu wahr evaluiert, falls es mindestens eine Belegung der Variablen gibt, für die der zu quantifizierende Ausdruck den effektiven booleschen Wert true liefert. Im Extremfall bedeutet dies, dass eine existenzielle Quantifizierung über eine leere Sequenz, die sich aus der Variablenbindung ergibt, grundsätzlich mit false bewertet wird. Der folgende Ausdruck wird entsprechend mit true bewertet, da mindestens eine Belegung der temporär gebundenen Variablen $x ("Lehner") die geforderte Zeichenkettenlänge erfüllt.

```
some $x in ("Lehner", "Schöning")
        satisfies fn:string-length($x) = 6
```

Die explizite existenzielle Quantifizierung spiegelt im Wesentlichen die Standardauswertung von Prädikaten innerhalb von FLWOR-Ausdrücken wider. Folgendes Beispiel illustriert dies, in dem alle Operationen im Sinne von Ärztefehlern ausgegeben werden sollen, bei denen der erste Einschnitt ohne Betäubung vorgenommen wurde:

```
<Arztfehler>
{
  for $o in fn:collection("Patienten")//Operation
  where some $i in $o//Einschnitt
                satisfies fn:empty($o//Anästhesie[. << $i])
  return $o
}
</Arztfehler>
```

Diese Anfrage ermittelt zunächst alle Operationen und bindet diese an die Variable $o. Die Operation wird jedoch nur in das Ergebnisdokument übernommen, falls die entsprechende existenzielle Quantifizierung zu true ausgewertet wird. In dem quantifizierenden Ausdruck werden alle Einschnitte innerhalb einer Behandlung dahingehend abgeprüft, ob – innerhalb derselben Behandlung – eine Betäubung existiert, die vor dem Einschnitt stattgefunden hat.

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8

Simulation durch where-Klausel oder Prädikat in Pfadausdruck

Die existenzielle Quantifizierung kann direkt durch die `where`-Klausel eines FLWOR-Ausdrucks bzw. als Filterausdruck in einem Pfadausdruck simuliert werden. Obiges Beispiel kann somit in analoger Weise wie folgt formuliert werden:

```
<Arztfehler>
{
  for $o in fn:collection("Patienten")
            //Operation[.//Einschnitt[not(../Anästhesie[1]<<.)]]
  return $o
}
</Arztfehler>
```

Die Variable `$o` wird dabei nur an die Operationen gebunden, bei denen die Reihenfolge »Einschnitt« vor »Anästhesie« innerhalb der Dokumentordnung vorgefunden wird. Der Pfadausdruck ist dabei in zwei Schritten zu interpretieren, die in Abbildung 6–1 verdeutlicht werden. Das Prädikat des äußeren Pfadausdrucks ist ein Pfadausdruck, der zunächst alle Einschnitte innerhalb der gerade betrachteten Operation identifiziert (und implizit auf deren Existenz prüft). Im zweiten Schritt erfolgt die Auswertung des Prädikates des inneren Pfadausdrucks, wobei das Prädikat nur dann zu `true` evaluiert wird, wenn der erste Anästhesieeintrag innerhalb dieser Operation nicht vor dem aktuell betrachteten Einschnitt (durch .-Notation gekennzeichnet) liegt.

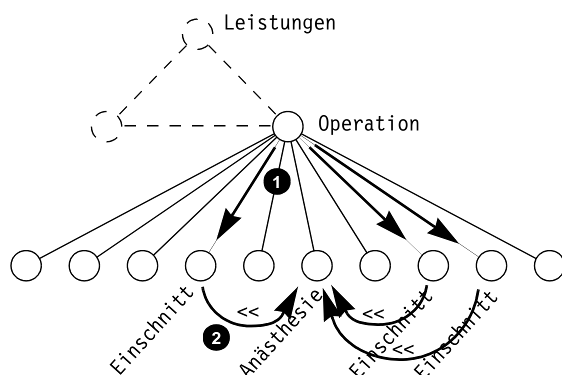

Abb. 6–1 *Simulation existenzieller Quantifizierung mit Pfadausdrücken*

Variablengültigkeit

Des Weiteren zeigt das obige Beispiel, dass die innerhalb eines quantifizierenden Ausdrucks gebundenen Variablen auch in dem zu überprüfenden Ausdruck nach dem `satisfies`-Schlüsselwort referenziert werden können. Eine Referenzierung außerhalb des quantifizierenden Ausdrucks ist hingegen nicht erlaubt, so dass folgende XQuery-Anwei-

sung zur Ausgabe der »kritischen« Behandlung mit dem entsprechenden zu früh durchgeführten Eingriff einen Fehler (zur Übersetzungszeit) liefern würde:

```
<Arztfehler>
{
  for $o in fn:collection("Patienten")//Operation
  where some $i in $o//Einschnitt
              satisfies fn:empty($o//Anästhesie[. << $i])
  return ($o, $i) (: Fehler, $i ist hier nicht bekannt :)
}
</Arztfehler>
```

Als Letztes sei angemerkt, dass existenzielle Quantifizierungen oftmals gewinnbringend in konditionale Ausdrücke eingebettet werden. So wird mit folgendem Ausdruck jeder Arzt mit einer Eigenschaft versehen, die darauf hinweist, ob ein Arztfehler vorgelegen hat oder ob bisher nur fehlerfreies Arbeiten aufgetreten ist:

```
for $a in fn:doc("Hochwaldklinik.xml")//Arzt
for $p in fn:collection("Patienten")//Patient_stationär
let $o := $p//Operation
where xqb:follow-xlink($p/Arzt/@xlink:href) is $a
return
  <Arzt>
  {
    $a/Name,
    <Anmerkung>
    { if some $i in $o//Einschnitt
                   satisfies fn:empty($o//Anästhesie[. << $i])
      then xs:string("Arztfehler liegt vor")
      else xs:string("Fehlerfreies Arbeiten")
    }
    </Anmerkung>}
  }
  </Arzt>
```

Universelle Quantifizierung

every ... in ... satisfies

Eine mit dem Schlüsselwort `every` eingeleitete universelle Quantifizierung liefert dann `true`, wenn für alle Belegungen der innerhalb des quantifizierenden Ausdrucks gebundenen Variablen der nach dem `satisfies`-Schlüsselwort stehende Ausdruck mit `true` evaluiert wird. Entsprechend ergibt nachfolgender Ausdruck `false`, da nicht alle Zeichenketten, die an `$x` gebunden werden, die geforderte Länge aufweisen.

```
every $x in ("Lehner", "Schöning")
        satisfies fn:string-length($x) = 6
```

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8

Im Gegensatz zur existenziellen Quantifizierung liefert eine universelle Quantifizierung true, falls die Variablen an eine leere Sequenz gebunden werden. Die Auswertung einer universellen Quantifizierung bricht mit dem Wert false ab, sobald eine Belegung gefunden worden ist, die den zu überprüfenden Ausdruck nicht befriedigt. Da die Strategie, nach welcher die Variablenbelegungen getestet werden, implementierungsabhängig ist, kann folgender Ausdruck entweder in false oder in einen Laufzeitfehler münden.

```
every $x in ("Lehner", "Schöning", 3.1416)
        satisfies fn:string-length($x) = 6
```

Im Fall einer existenziellen Quantifizierung wäre das Ergebnis für obigen Ausdruck ebenfalls offen und kann zwischen true und einem Laufzeitfehler variieren.

Die universelle Quantifizierung wird im Anwendungsbeispiel an der Aufgabe illustriert, alle Patienten zu finden, die bisher nur mit Morphium behandelt wurden und für die sich somit der Verdacht auf eine Medikamentenabhängigkeit ergibt:

```
<MedikamentenabhängigePatienten>
{
  let $m := fn:doc("Verbrauchsartikel.xml")//
                              Artikel[Bezeichnung = "Morphium"]
  for $p in fn:collection("Patienten")/*
  where every $a in $p//verbrauchter_Artikel
                satisfies ($a/@Artikel_id = $m/@ID)
  return
    <Patient>{ $p/Name/Nachname, $p/Name/Vorname }</Patient>
}
</MedikamentenabhängigePatienten>
```

Dazu werden zunächst Morphium-Medikamente an die Variable $m gebunden. Im universell quantifizierenden Ausdruck wird anschließend überprüft, ob jedes für den aktuell betrachteten Patienten verbrauchte Medikament mit einem Morphium-Medikament korrespondiert. In einem entsprechenden Fall wird die where-Bedingung mit true ausgewertet und der Patientenname ausgegeben.

Quantifizierende Ausdrücke mit mehreren Variablen

Wie der Auszug aus der Grammatik für quantifizierende Ausdrücke zu Beginn dieses Abschnitts zeigt, können mehrere Variablen temporär für den jeweiligen Ausdruck gleichzeitig gebunden werden. Die Semantik, die sich dahinter verbirgt, besteht darin, dass alle Kombinationen von Variablenbelegungen für den Test des zu befriedigenden

Ausdrucks herangezogen werden. Im Fall universeller Quantifizierung müssen demnach alle möglichen Belegungen den zu überprüfenden Ausdruck zu true evaluieren; bei existenzieller Quantifizierung muss mindestens eine Kombination möglicher Variablenbelegungen gefunden werden, die den zu befriedigenden Ausdruck zu true evaluiert.

```
(: Ausdruck 1 :) some $x in (1,2,3), $y in (2,3,4)
                           satisfies $x*$y = 6

(: Ausdruck 2 :) every $x in (1,2,3), $y in (2,3,4)
                            satisfies $x*$y = 6
```

Während der erste Ausdruck zu true evaluiert wird, da es mindestens eine Kombination der Belegung der beiden Variablen $x und $y ($x=2, $y=3 bzw. $x=3, $y=2) gibt, die die geforderte Gleichung erfüllt, trifft dies für den zweiten Ausdruck nicht zu, da mindestens (irgend-) eine Kombination die Gleichung nicht erfüllt.

Explizite Typzusicherung

Darüber hinaus erlaubt die Bindung von Variablen in quantifizierenden Ausdrücken in Analogie zur Bindung von Variablen in let- und for-Klauseln eines FLWOR-Ausdrucks eine explizite Typzusicherung durch das Schlüsselwort as (Abschnitt 3.8.3). Der erste Ausdruck würde in einem Typfehler resultieren, da die Elemente der Sequenz nicht dem angegebenen Typ xs:integer entsprechen. Der zweite Ausdruck hingegen würde true ergeben, da nach der expliziten Konvertierung numerischer Werte in eine ganze Zahl (Abschnitt 3.2.4) der Ausdruck $x*2=6 durch das erste Element in der Sequenz erfüllt sein würde.

```
(: Ausdruck 1 :) some $x as xs:integer in (3.141592654, 2.718281828)
                                         satisfies $x*2 = 6
(: Ausdruck 2 :) some $x as xs:integer in (xs:integer(3.141592654),
                                             xs:integer(2.718281828))
                                          satisfies $x*2 = 6
```

Im Fall einer Konvertierungsverletzung ist die Generierung eines Laufzeitfehlers implementierungsabhängig. Zum Beispiel könnten die beiden folgenden Anweisungen in einen Laufzeitfehler münden oder true bzw. false liefern.

```
some $x as xs:integer in (3, "Lehner", "Schöning")
                     satisfies $x*2 = 6

every $x as xs:integer in (2, "Lehner", "Schöning")
                     satisfies $x*2 = 6
```

Sofern der XQuery-Prozessor das Konzept der statischen Typprüfung (Kapitel 8) implementiert hat, können entsprechende Fehler bereits zur Übersetzungszeit gefunden werden.

Wolfgang Lehner / Harald Schöning, XQuery, dpunkt.verlag, ISBN 978-3-89864-266-8