FOXpath navigation of physical, virtual and literal file systems

Hans-Jürgen Rennau, Traveltainment GmbH <`hrennau@yahoo.de`>

Abstract

The FOXpath language extends the XPath language by adding support for file system navigation. This paper explores possibilities how to extend file system navigation beyond physical file systems and include logical file systems like jar files, SVN repositories or github projects. The extension is based on a set of simple concepts related to URIs and their processing, and it is implemented as a FOXpath processor which supports the navigation of physical and various types of logical file systems.

Table of Contents

Introduction

Tree-structured information is ubiquitous. A simple shopping list, for instance, which groups items by department or shop, is a tree. Other examples include file systems, jar files, SVN repositories, NOSQL databases, github projects and Java packages. Major advantages of a tree structure are obvious. It favours a clear understanding where to find what and enables an intuitive addressing of single items. It invites to dwell on a chosen level of detail or abstraction. It domesticates complexity and appears natural to human thought. One particular advantage, however, is still well-hidden from broad attention: this is an amazing **conductivity** which information attains when arranged as a tree. The term borrowed from physics should capture the ease of traversing a bulk of information, in order to arrive at points of interest. This quality is revealed by XPath (6), an expression language for navigating the contents of XML node trees. Consider this example:

```
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
```

The expression returns the departure airports of evening flights to the airport JFK. In order to arrive at the `airport` items of interest, complex information is traversed in a single fell sweep.

The elegance of such expressions is not based on specific qualities of the XML data format – it reflects inherent qualities of tree-structured information. Consider the key concepts of XPath: the combination of *navigation axis*, *node test* and *predicates* into a step; the chaining of steps into a *path*. The XPath model of navigation is not tied to XML and would perhaps make sense in many tree-structured collections of information. How about file systems? Like XML documents, they are trees, and finding particular folders and files is as important (at least) as finding particular XML elements and attributes. Can we have something like XPath for file systems?

The FOXpath language (2) turns this idea into reality. It is a superset of XPath 3.0, adding file system navigation – multi-step movement across the hierarchical structure of the file system. Consider this FOXpath expression:

```
\projects\gateway\\routes-*.xml[not(ancestor~::blacklisted)]
```

Its value is the set of documents about flight routes found in the gateway project, but not at any depth under a `blacklisted` folder. The *integration* of XPath-like file system navigation into XPath has two interesting benefits. First, file system navigation can be seamlessly combined with file content navigation – even within a single expression. Example:

```
\projects\gateway\\routes-*.xml[not(ancestor~::blacklisted)]
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
```

Here the navigation to file resources (\ seperated steps) is continued by navigation *within* them (/ seperated steps). The next example,

```
\projects\gateway\\routes-*.xml[not(ancestor~::test)]
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
=> distinct-values() => sort()
```

demonstrates the advantages of navigation embedded into a full-blown expression language: navigation can be used as operand of other operations which have nothing to do with navigation.

FOXpath thus creates an interesting experience of information: a tree (file system) whose leaf nodes (files) are themselves trees (XML node trees), merged into a continuous space of information by a single navigation model, which is embedded into a fully composable expression language.

This paper reports my efforts to push things one step further: to extend file system navigation beyond physical file systems, into the realm of logical file systems like jar files or github projects (3). As a preparatory step I provide some information about the FOXpath language.

The FOXpath language

FOXpath (2) is a superset of the XPath 3.0 language (6). The purpose of extension is adding support for file system navigation which has the same look and feel as node tree navigation. The extension of XPath can be summarized as follows:

- Addition of a new expression kind, the *URI axis step* (e.g. `descendant~::foo`)
- Addition of a new operand, the *URI path operator* (e.g. `foo\bar`)
- Semantic extensions (backward-compatible)

A *URI axis step* expression mirrors the axis step expression of XPath, being a combination of navigation axis, name test and optional predicates. A URI axis step maps input URIs to output URIs related to the input URIs by a navigation axis (child, descendent, parent, ancestor, …). File system navigation axes are defined in analogy to the navigation axes defined by the XPath language.

The *URI path operator* chains two consecutive steps of URI navigation into a complex operation, echoing the way how the path operator of XPath chains two consecutive steps of node navigation.

Semantic extensions modify the semantics of a few XPath expressions in such a way that

- Any expression which yields a value (rather than raises an error) when evaluated as an XPath expression yields the same value when evaluated as a FOXpath expression
- Some expressions which raise an error when evaluated as an XPath expression yield a value when evaluated as a FOXpath expression

The semantic extensions enable a seamless integration of file system and node navigation. Consider the expression:

```
\projects\\foo.xml//bar
```

which navigates to `foo.xml` files and then to the `bar` elements which they contain: without semantic extension, the node navigation would raise an error as the context item is a resource URI, not a node. However, the extended semantics of node axis steps prescribe automatic conversion of an atomic context item into a node by parsing the document found at the document URI given by the context item. Further semantic extensions concern the definition of the effective boolean value, which in certain cases yields a value in FOXpath, whereas an error is raised in XPath.

Besides file system navigation, the FOXpath language adds to XPath 3.0 a few more extensions:

- Support for external variables *(as defined by XQuery)*
- Support for namespace declarations *(as defined by XQuery)*
- Support for simple FLWOR expressions, containing one or more `let` clauses and/or one or more `for` clauses *(simplified version of XQuery's FLWOR expression)*
- Support for the `arrow` operator *(as defined by XPath 3.1)*
- About 60 extension functions

It is important to note, however, that FOXpath does not extend or modify the data model of XPath (XDM, (8). File system navigation is viewed as operations applied to strings interpreted as URIs.

To summarize, the FOXpath language complements node tree navigation with file system navigation, moving around in a tree of folders and files. The FOXpath model of file system navigation is a faithful copy of the XPath model of node navigation. In particular, navigation is viewed as a sequence of steps which typically combine a navigation axis (child, descendant, parent, ancestor, …) with an item test (name test) and optional predicates.

The challenge of generalization

Many systems of resources appear to be organized like file systems, although they are not implemented by a physical file system. Examples include the contents of archive files, the repositories of version control systems, some NOSQL database systems, project repositories like github and the resources exposed by many RESTful web services. If the scope of FOXpath navigation could be extended to include such logical file systems, the usefulness of the FOXpath language might increase considerably.

The model of FOXpath navigation is based on navigation axes. These presuppose tree-structured collections of URIs, of which physical file systems are just a special case. The FOXpath language is therefore in no way tied to physical file systems, and navigation of non-physical and physical file systems should be equally possible. This situation suggests a dual challenge:

1. To provide a *conceptual framework* relating the abstract navigational functionality of the FOXpath language to an implementation-defined set of logical file systems
2. To provide a proof of concept *implementation* of the FOXpath language which supports the navigation of physical as well as various kinds of non-physical file systems.

Concepts

This section sketches a conceptual framework relating the navigation model of the FOXpath language – based on navigation axes applicable to resource URIs – and actual systems of resources to which such navigation may be applied. Formal rigour and precision are sacrificed to readability. As an example of this imprecision, the contained-by relationship between a given URI and a file system is not discussed, although it is important for the exact meaning of an operation like "mapping a URI to its child URIs", as such a mapping can only be unambiguously defined in the context of an actual file system.

File system types

Desiring a generalization of FOXpath navigation, a good starting point is a generalized concept of file systems. Loosely speaking, a *logical file system* is a tree of URIs which has a single root, whose leaf URIs can be resolved to a chunk of content (sequence of characters or bytes) and whose inner nodes *cannot* be resolved to content, but are perceived as containers of other URIs. The generalized notions of files and folders are thus the outer and inner nodes of such a tree of URIs.

As the notion of a "tree of URIs" is vague, we introduce the concept of **child URI**. A URI U2 is said to be a child URI of another URI U1 if U1 is a prefix of U2, followed by a slash and a non-empty sequence of characters not containing a slash. So, for example, the URI

```
https://github.com/marklogic/semantic
```

is a child URI of

```
https://github.com/marklogic
```

A logical file system can thus be characterized as a tree of abstract nodes representing URIs, where child nodes are child URIs and only leaf nodes are URIs pointing to content. Some examples:

- The contents of a jar file (or some other archive file)
- The contents of a BaseX database (or some other NOSQL database)
- The contents of an SVN repository (or some other version control system)
- The contents of a github project
- The URIs exposed by a typical RESTful web service

A logical file system is usually backed by APIs enabling some form of navigation and content retrieval. Applications based on such APIs may create the semblance of a file system that can be browsed in the usual way. As an example, consider the github web service (4) and the github web site (3). When APIs and applications create the semblance of a physical file system, we speak of a **virtual file system**.

While virtual file systems *behave* like a tree of URIs, they usually do not expose any literal representation of the tree and its nodes, a representation composed of distinct units of information describing. individual URIs. Such a literal representation might for example be an XML document or an RDF graph, representing URIs by XML elements or RDF nodes, respectively. Note however that for any logical file system such a representation can be *constructed* by tools using the available APIs. We propose the notion of a **literal file system** which is a tree of resource URIs described by an artifact composed of distinct units representing single URIs. A literal file system may, but need not be a one-to-one representation of a physical or virtual file system. It may contain selected parts of a single or multiple logical file systems, and also URIs not associated with any file system at all. Such unconstrained compositions can be merged into a logical file system (a tree of URIs) because the URIs actually *exposed* need not be the original URIs. The URIs integrated into a literal file system are *navigation URIs*, which are associated with *resource access URIs* – the original URIs which may be equal to or different from the navigation URI. A literal file system can be thought of as a special kind of catalog, mapping a set of URIs (resource access URIs) to a tree-structured set of target URIs (navigation URIs) and providing additional information about the resources, like file size or date of last modification).

In summary, we introduce the notion of a **logical file system** which may be a **physical file system**, a **virtual file system** or a **literal file system**. The FOXpath language is tied to the concept of a logical file system, rather than a physical file system. In principle, it should be possible to support navigation of all kinds of logical file system. To realize this potential, it is important to identify key operations which (a) serve as an abstraction hiding the actual type of file system at hand, (b) serve as building blocks from which the complete functionality of FOXpath navigation can be constructed.

URI operations

URI operations are operations applied to URIs and yielding a result which is valuable in the context of file system navigation and the retrieval of resource contents and resource properties. We reserve the term for basic operations, usable as building blocks from which to construct more complex operations. A crucial aspect is to define these operations in a generic way which is independent of the type of logical file system at hand (physical file system, archive file, …). For instance, an SVN command defined to map the URI of an SVN folder to the URIs of contained SVN folders and files is not a generic URI operation; an operation defined to map a URI to its child URIs is a generic URI operation. Think of URI operations as a basic interface exposed by a logical file system of any type.

Navigation primitives

File system navigation as defined by the FOXpath language is based on navigation axes. These are mappings of an input URI to related URIs, e.g. descendant or ancestor URIs. Upward navigation is a purely syntactical operation (the removal of trailing URI steps) and downward navigation can be achieved by single or recursive mapping of a URI to its child URIs. Therefore the complete implementation of all FOXpath-defined axes requires only a single navigation primitive, which is a mapping of an input URI to its child URIs.

Practical considerations, however, led me to define two navigation primitives designed with the goal of balancing simplicity and the efficiency of operations based on these primitives:

```
uri-to-child-uris      ($uri, $nameFilter?, $kindFilter?) as xs:anyURI*
uri-to-descendant-uris ($uri, $nameFilter?, $kindFilter?) as xs:anyURI*
```

where `$uri` is the input URI, `$nameFilter` is an optional name pattern to be matched by the trailing step of the returned URIs, and `$kindFilter` is an optional filter retaining only files or only folders.

Content retrieval

Files are URIs pointing to content, which is a sequence of characters or bytes. We define the following URI operations for content retrieval:

```
file-text       ($uri, $encoding) as xs:string
file-bytes      ($uri) as xs:base64Binary
file-doc        ($uri) as document-node()
file-json-doc   ($uri) as document-node()
```

Resource property retrieval

A file system resource has a few basic properties:

- resource kind, which is `file` or `folder`
- timestamp of last modification
- size in bytes (only if the resource is a file, not a folder)

The following URI operations can be used to retrieve the property values:

```
resource-kind($uri) as xs:string  // value is one of 'file' or 'folder'
resource-date($uri) as xs:dateTime?
```

```
resource-size($uri) as xs:nonNegativeInteger?
```

URI processor

A **URI processor** is a program or program module which implements the URI operations for URIs stemming from a particular type of logical file system. URI processors are therefore classified by the type of logical file system they can handle. A few examples:

- file system URI processor
- archive file URI processor
- BaseX database URI processor
- SVN URI processor
- UTREE processor
- UGRAPH processor

UTREE and UGRAPH processors deal with literal file systems defined by XML documents and RDF graphs, respectively. See [the section called "Literal file systems"] for details.

URI dispatchal rules

A FOXpath implementation including a particular type of URI processor may support FOXpath navigation of the corresponding type of file systems. An SVN URI processor, for example, may enable FOXpath navigation of SVN repositories.

Navigation support for *multiple* types of file system requires an identification of the type of file system to which a given URI belongs. This operation I call **URI dispatchal**, as it amounts to dispatching a URI to the appropriate URI processor. The dispatchal rules of a FOXpath implementation are implementation defined. Obvious possibilities are based on URI schemes and URI prefixes. Rules may reference configuration data, in which case they must define the format and semantics of such data.

Note that dispatchal rules do not identify an *instance* of a file system – e.g. a particular SVN repository. They only identify the *type* of file system.

Literal file systems

A literal file system is an association between a collection of resources and a tree of resource URIs. The tree facilitates discovery of the resources via navigation, and it enables retrieval of resource contents and basic resource properties. The tree should be viewed as a tree of logical components, to be distinguished from data encoding the tree. I have defined two data formats, where one is an XML document and another is an RDF graph.

Logical model

A literal file system is an instance of an information model designed to model a system of folders and files. It is a tree of abstract nodes defined and constrained as follows:

- Every leaf node represents either a file or an empty folder
- Every inner node represents a non-empty folder
- A file node has the following properties:
  - A navigation URI
  - A resource access URI
  - File size (optional)
  - Timestamp of last modification (optional)
- A folder node has the following properties:
  - A navigation URI
  - Timestamp of last modification (optional)

- For any two nodes N1 and N2 where N2 is a child node of N1, the navigation URI of N2 is a child URI of the navigation URI of N1

Navigation and resource access URIs are distinct properties of a file node – they may, but need not be equal. A navigation URI is a URI associated with a resource in order to make it conveniently discoverable. From the file system user's point of view, the navigation URI is also the URI which is resolved to the resource contents. From the file system implementation's point of view, however, resource access is not provided by the navigation URI, but by the associated resource access URI. The tree structure emerging from the collected navigation URIs does in no way constrain the locations of the resources represented by that tree, nor the protocols used in order to retrieve those resources. The literal file system *may* be a one-to-one representation of some physical or virtual file system. It may also be a projection of such a system, representing some and not representing other resources contained by the original system. A literal file system may also represent a collection of resources belonging to *several* logical file systems, or even not belonging to any known file system at all.

Data formats - UTREE and UGRAPH

Two data formats representing a literal file system have been defined: an XML based representation (UTREE) and an RDF based representation (UGRAPH).

The XML format (UTREE)

UTREE is an XML format for representing the contents of one or several literal file systems. Element summary:

- Document root element: `<trees>`
- File system root element: `<tree>`
- Inner nodes: `<dir>` elements
- Leaf nodes: `<file>` and/or `<dir>` elements

Attribute summary:

- **On <trees>**
  - @uriPrefix – optional; if used, a URI processor must treat any URI starting with that prefix as either described by a <tree>, <file> or <dir> element in that document or not corresponding to any existent resource

- **On <tree>**
  - @baseURI – the navigation URI of the literal file system root, followed by a slash, unless the navigation URI ends with a slash

- **On <file>**
  - @name – the file name
  - @path – the trailing part of the navigation URI; the URI is obtained by appending the attribute value to the @baseURI value of the containing <tree> element
  - @accessURI – the URI to be used for resource access
  - @lastModified – timestamp of last modification (optional, may be unknown)
  - @size – size of the file, in bytes (optional, may be unknown)

- **On <dir>**
  - @name – the folder name
  - @path – the trailing part of the navigation URI; the navigation URI is obtained by appending the attribute value to the @baseURI value of the containing <tree> element
  - @lastModified – timestamp of last modification (optional, may be unknown)

Constraints

- A child node of a given node N has a navigation URI which is a child URI of the navigation URI of N

The RDF format (UGRAPH)

UGRAPH is an RDF format for representing the contents of a literal file system. In the following description, the prefix `fs:` represents the URI

`http://www.foxpath.org/ns/rdf/filesystem/`

In an instance of UGRAPH, two types (`rdf:type`s) of resources are distinguished:

- `fs:dir` – resource represents a folder
- `fs:file` – resource represents a file

Properties of `fs:dir` resources:

- `fs:navURI` – the navigation URI of a folder
- `fs:parentDir` – resource URI of the parent folder resource (optional, as the root folder has no parent folder)
- `fs:name` – the folder name (optional, as the root folder may have no name)
- `fs:lastModified` – timestamp of last modification (optional, may be unknown)

Properties of `fs:file` resources:

- `fs:navURI` – the navigation URI of a file
- `fs:accessURI` – the resource access URI of a file
- `fs:parentDir` – resource URI of the parent folder resource
- `fs:name` – the file name
- `fs:lastModified` – timestamp of last modification (optional, may be unknown)
- `fs:fileSize` - size of the file, in bytes (optional, may be unknown)

Constraints:

- An UGRAPH has exactly one `fs:dir` resource without `fs:parentDir` property; this resource represents the root folder of the system
- An `fs:dir` resource without `fs:parentDir` property must not be a descendant URI of a resource contained by the graph – any non-root resource is connected to the root folder by a chain of `fs:parentDir` properties
- If a resource R has an `fs:parentDir` D, D is also contained by the UGRAPH
- If a resource R has an `fs:parentDir` D, the `f:navURI` of R is a child URI of the `f:navURI` of D
- A non-root folder MUST have a name

Implementation

This section describes the implementation of a FOXpath language processor which supports the navigation of physical, virtual and literal file systems. The language implementation is accompanied by a command-line tool for creating literal file systems represented by UTREE documents or UGRAPH triple sets.

Overview

A FOXpath language processor can be downloaded from here:

`https://github.com/hrennau/foxpath`

The scope of file system navigation includes:

- physical file systems
- archive files (jar, zip, …)
- BaseX databases

- SVN repositories
- github.com repositories (represented by UTREE or UGRAPH)
- UTREE literal file systems
- UGRAPH literal file systems

The processor ships with a command-line tool for creating literal file systems (UTREE or UGRAPH style) ...

- from SVN repositories
- from github.com repositories

The FOXpath processor is implemented as a set of XQuery modules, XQuery version 3.1. As several extension functions of the XQuery processor BaseX (1) are used, FOXpath can currently only be executed using the XQuery processor BaseX. The implementation of the FOXpath language is a set of ordinary XQuery modules – it does not involve any changes of the XQuery processor.

The *XQuery interface* of the processor is a function resolving FOXpath expressions to their value.

A *command-line interface* is provided by shell scripts (`fox.bat`, `fox.sh`). They resolve FOXpath expressions provided either as a command-line parameter or as file contents.

Interfaces

The *XQuery interface* of the implementation is an XQuery function resolving FOXpath expressions to their value:

```
declare function f:resolveFoxpath(
  $foxpath as xs:string,    (: the expression text :)
  $ebvMode as xs:boolean?,  (: true => return effective boolean value :)
  $context as xs:string?,   (: folder URI providing file system context :)
  $options as map(*)?,      (: UTREE locations, UGRAPH endpoints, ... :)
  $externalVariableBindings as map(xs:QName, item()*)?)
      as item()*            (: the expression value :)
```

The *command-line interface* is a shell script (`fox.bat`, `fox.sh`) resolving FOXpath expressions to their value:

```
fox [-t utree-dir] [-g ugraph-endpoint] [-v name=value]* expression-text
fox [-t utree-dir] [-g ugraph-endpoint] [-v name=value]* -f expression-file
```

Implementation details

The XQuery-based implementation of the FOXpath language relies on several *XQuery extension functions* supported by the XQuery processor BaseX (1). The following tree-structured representation summarizes the dependencies of functional areas. Underlying view:

- FOXpath = XPath + file system navigation
- file system = physical file system | archive file | BaseX database | SVN repo | UTREE | UGRAPH

Leaf nodes of the tree are XQuery extension functions, inner nodes are areas of functionality, the root node is the FOXpath language, as implemented. The notations [expath] and [basex] mark functions as defined either by EXPath (2) or BaseX (1).

```
FOXpath
.  XPath
.  .  [basex] xquery:eval                // for partial function applications
.  logical_file_system_navigation
.  .  physical_file_system
.  .  .  [expath] file:list              // downward navigation
.  .  .  [expath] file:is-dir            // retrieval of file properties
.  .  .  [expath] file:last-modified     // retrieval of file properties
.  .  .  [expath] file:size              // retrieval of file properties
.  .  .  [expath] file:read-binary       // resource retrieval
.  .  archive_file
.  .  .  [expath] file:read-binary       // downward navigation
.  .  .  [basex]  archive:entries        // downward navigation
.  .  .  [basex]  archive:extract-text   // resource retrieval
.  .  .  [basex]  archive:extract-binary // resource retrieval
.  .  basex_database
.  .  .  [basex]  db:list                // downward navigation
.  .  svn
.  .  .  [basex] proc:system             // access to SVN CL interface
.  .  utree
.  .  utree_for_github
.  .  .  [expath] http:send-request      // access to github REST API
.  .  .  [basex]  convert:binary-to-string// github API delivers binary
.  .  ugraph
.  .  .  [expath] http:send-request      // access to SPARQL endpoints
.  .  ugraph_for_github
.  .  .  [expath] http:send-request      // access to github REST API
.  .  .  [basex] convert:binary-to-string // github API delivers binary
```

URI dispatchal

URI dispatchal maps a given URI to at most one of the supported file system types. URI dispatchal rules are *implementation defined*, and they are crucial for supporting the navigation of multiple file system types.

URI dispatchal configuration

Dispatchal is controlled by static rules and configuration data passed to the expression resolver at runtime. The configuration data consist of

- UTREE folders – folders containing one or more UTREE documents
- UGRAPH endpoints - SPARQL endpoints exposing one or more UGRAPH triple sets

UTREE folders and UGRAPH endpoints are passed as parameters to the FOXpath processor. On the command line, the UTREE folders and UGRAPH endpoints are specified using option –t and –g, respectively:

```
fox –t "utree-dirs" ...
fox –g "ugraph-endpoints" ...
```

On the XQuery interface of FOXpath, UTREE folders and UGRAPH endpoints can be specified as entries in a map of options (keys UTREE_DIRS and UGRAPH_ENDPOINTS).

URI dispatchal rules

The following URI dispatchal rules are evaluated in order, and the first rule inferring a file system type provides the final result.

Rule #1 - archive rule

A URI containing one or more steps with the text `#archive#` points to contents of an archive file. The *archive file URI* is given by the path prefix preceding the last occurrence of an `#archive#` step. The *within-archive path* is given by the path suffix following the last occurrence of an `#archive#` step.

For example, the URI

```
/apache-jena-3.1.0//jena-tdb-3.1.0.jar/#archive#/org//*.properties
```

references all resources found in the archive file

```
/apache-jena-3.1.0//jena-tdb-3.1.0.jar
```

at locations matching

```
/org//*.properties
```

URI dispatchal rules are applied recursively in order to access the archive file. The archive file may therefore be contained by any of the file system types supported by the FOXpath processor. It may, for example, itself be located in an archive file.

Rule #2 - literal file system rule

If the URI does not refer to archive contents, run time inspection of the provided UTREE documents and UGRAPH triples enables the FOXpath processor to infer a mapping of URI prefixes to UTREE documents and UGRAPH endpoints:

- A URI with one of the UTREE-associated prefixes is inferred to belong to a UTREE-based literal file system
- A URI with one of the UGRAPH-associated prefixes is inferred to belong to a UGRAPH-based literal file system

Rule #3 - BaseX rule

If the URI starts with `basex:/`, the URI refers to a resource contained by a BaseX database.

Rule #4 - SVN rule

If the URI scheme starts with `svn-` (e.g. the URI starts with `svn-file:/`, `svn-http:/` or `svn-https:/`), the URI refers to a resource contained by an SVN repository.

Rule #5 - github rule

If the URI starts with `https://api.github.com/repos/`, the URI refers to a resource contained by a github project.

Rule #6 - physical file system rule

- If URI starts with `file:/`, the URI refers to a physical file system resource
- If the URI has no URI scheme, the URI refers to a physical file system resource

Rule #7 - match failure rule

A URI for which no file system can be determined, does not belong to a (recognized) file system. This implies:

- Attempts at downward navigation (e.g. navigation to child URIs) yield the empty sequence
- Attempts to retrieve file properties yield the empty sequence
- The semantics of content retrieval (functions `fn:doc`, `fn:unparsed-text`, ...) are not affected - content retrieval is controlled by the URI scheme

Proprietary URI schemes

Note the use of **proprietary URI schemes**

- `svn-…/` (… = file | http | https)

- `basex:/`

A *proprietary URI* is derived from a *standard URI* by inserting a prefix (`svn-`, `basex:`) or an intermediate step (`#archive#`), which indicates the type of containing file system. The FOXpath implementation relies on the use of proprietary URIs in order to recognize certain types of file system, like SVN, BaseX or archive, which cannot be inferred from the URI scheme. Proprietary URIs are used by the FOXpath processor, but never passed to the underlying APIs for navigation and retrieval. Note however that the *choice* of underlying API is crucial, as navigation, for instance, must be handled differently according to the file system type. The switching between proprietary and standard URIs is transparent to the FOXpath user, though, who supplies and receives only the proprietary URIs. Consider the following examples:

```
svn-https://svn.apache.org/repos/asf/tomcat/jk/trunk/*
svn-https://svn.apache.org/repos/asf/tomcat/jk/trunk/KEYS/grep('apache')
https://svn.apache.org/repos/asf/tomcat/jk/trunk/*
```

The first expression yields a list of child URIs, which of course have the same URI scheme as the parent URI. These URIs cannot be used outside of FOXpath expressions. Within FOXpath expressions, however, they are used as if they were standard URIs, as the second expression demonstrates, which retrieves resource contents. The third expression will probably return the empty sequence, as the URI is not recognized as referring to SVN resources.

Tool support for constructing literal file systems

The creation of literal file systems is supported by a command-line tool called `lifis` (literal file system). The tool enables the convenient construction of literal file systems whose structure and contents capture the structure and contents of a physical or virtual file system – e.g. a github or SVN repository. The literal file system can be restricted to a fragment of the original system (rooted in a non-root folder) and to the result of filtering the original system contents (ignoring selected folders or files). The tool user identifies one or several source file systems and the literal file system format (UTREE or UGRAPH). The tool creates a UTREE document or a UGRAPH graph. Additional options can be specified which load the UTREE document into a BaseX database or load the UGRAPH triples into a TDB triple store (5).

Examples

Several examples will demonstrate FOXpath navigation of various types of logical file systems. The examples are provided as calls of the `fox` script, a command-line tool for resolving FOXpath expressions. The examples use a syntax variant of the FOXpath language called **friendly syntax**, in which the roles of slash and backslash are swapped. The *axis step operator* inherited from XPath is represented by a *backslash*, not by a slash as in XPath. The *slash* represents the URI axis step operator, which separates file system navigation steps.

Navigating the physical file system

We explore an installation of the application server Wildfly (version 10.1.0). To get started, we list the top-level folders, together with the number of contained XML files:

```
fox "/wildfly101/*[is-dir()]
     /concat(rpad(file-name(), 25, '.'), '   ', count(.//*.xml))"

=>
.installation ..........   0
appclient ..............   1
bin ....................   1
docs ...................   15
domain .................   4
modules ................   362
```

```
standalone .............   4
welcome-content .........   0
```

Next, we list the paths of all XML files not contained in the `modules` or the `docs` folder:

```
fox "/wildfly101//*.xml[not(ancestor~::modules)][not(ancestor~::docs)]"

=>
/wildfly101/appclient/configuration/appclient.xml
/wildfly101/bin/jboss-cli.xml
/wildfly101/domain/configuration/domain.xml
…
```

Finally, we list all XML files which are not described by any of the (278) XSDs contained in the installation:

```
fox "let $xsdnames :=
           /wildfly101//*.xsd
           \xs:schema\xs:element\@name\QName(..\..\@targetNamespace, .)
     return /wildfly101//*.xml[not(node-name(\*) = $xsdnames)]"

=>
/wildfly101/docs/licenses/licenses.xml
/wildfly101/modules/system/layers/base/org/jboss/genericjms/main/META-INF/ra.xm
```

Navigating the contents of an archive file

We navigate into a .jar file and extract a .properties file defining messages. The .jar file has a name pattern “mod_cluster*”. The .properties file is found by looking for the string “# error messages”:

```
fox "/wildfly101//mod_cluster*.jar
     /#archive#//*.properties[grep('# error messages')]/file-content()"

=>
# Regular messages
modcluster.advertise.start=Listening to proxy advertisements on {0}:{1}
modcluster.context.disable=Undeploy context [{0}] from host [{1}]
modcluster.context.enable=Deploy context [{0}] to host [{1}]
…
```

Navigating the contents of BaseX databases

We list the names of all BaseX databases containing XSD files:

```
fox "basex://*[.//*.xsd]"
```

Navigating an SVN repository

We access the public SVN repository of the Apache foundation and extract the names of developers involved in the xerces project:

```
fox "svn-https://svn.apache.org/repos/asf
     /xerces/xml-commons/trunk/status.xml
     \\developers\person\@name
     => string-join(', ')"
=>
Shane Curcuru, David Crossley, Neil Graham, Ilene Seelemann, Norman Walsh,
Michael Glavassevich, Morris Kwan, Jeremias Maerki, Cameron McCormack,
Volunteer needed
```

Navigating a UTREE file system

In a preparatory step, we create a UTREE file system of the github.com projects published by the organisation MarkLogic. We use the `lifis` tool:

```
lifis "github?org=marklogic,format=utree" > /utree/github/ml/utree-ml.xml
```

Now we can navigate this system, specifying the UTREE folder via `fox` option –t. We create a report of dependencies defined by any of the POM files in any of the MarkLogic projects:

```
fox -t /utree/github/ml
  "https://github.com/marklogic//pom.xml
   \\*:dependency
   \concat(rpad(*:groupId, 35, '.'), ' ',
           rpad(*:artifactId, 30, '.'), ' ',
           *:version)
   => distinct-values() => sort()"

=>
...
log4j ............................ log4j ....................... 1.2.17
net.sf.opencsv ................... opencsv ..................... 2.3
net.sourceforge.htmlcleaner ....... htmlcleaner ................. 2.4
net.sourceforge.openutils ......... openutils-log4j .............
org.apache.avro .................. avro-tools .................. 1.7.4
org.apache.commons ............... commons-csv ................. 1.2
org.apache.derby ................. derby .......................
org.apache.hadoop ................ hadoop-annotations .......... 2.6.0
...
```

In order to improve performance, we load the document into a BaseX database which we call `utreebase`. We repeat our reporting, now specifying the UTREE *database*, rather than the UTREE folder:

```
fox -t basex://utreebase "https://github.com/marklogic//pom.xml …"
```

Navigating a UGRAPH file system

In the previous section we created a literal file system of the UTREE type in order to navigate the github projects of MarkLogic. We may alternatively create a literal file system of the UGRAPH type:

```
lifis "github?org=marklogic,format=ugraph" > /ugraph/github/ml/ugraph-ml.ttl
```

After loading the triples into a TDB database (5):

```
tdbloader --loc=/tdb/ugraph-github-ml /ugraph/github/ml/ugraph-ml.ttl
```

we start a Fuseki server (5), which exposes the database as a SPARQL endpoint:

```
fuseki-server --loc=/tdb/ugraph-github-ml /marklogic
```

We repeat our reporting step, now specifying the UGRAPH endpoint via option -g:

```
fox -g http://localhost:3030/marklogic
  "https://github.com/marklogic//pom.xml
   \\*:dependency
   \concat(rpad(*:groupId, 35, '.'), ' ',
           rpad(*:artifactId, 30, '.'), ' ',
           *:version)
   => distinct-values() => sort()"
```

Discussion

The FOXpath language extends the XPath language by adding support for file system navigation. Navigation is modelled in terms of new expressions which map a URI to other URIs, based on structural relationships within a tree of URIs (e.g. asserting a URI to be a descendant or the parent of another URI). Expression semantics make no assumptions about the rationale of these relationships. In particular, they may be established by coexistence within a physical file system, but also by coexistence within a virtual file system, which attains the appearance of a file system due to APIs or applications. Literal file systems, finally, enable a complete decoupling of the original resource URIs and the URIs used for navigation, so that the relationships driving navigation can be constructed independently of coexistence and relationships within a physical or virtual file system.

What has been called "file system navigation" throughout this paper might therefore more aptly be called URI navigation. The generic nature of this added functionality points to a certain lack of concepts as how to bind potential use to actual, implementation-defined uses. This paper should be seen as a tentative step towards closing the gap and also towards proving the practical value of the new potential.

URI navigation is about discovering resources of interest. The discovery is based on relationships between URIs, which are not established by explicit hyperlinks. The relationships are implied by the structure of the URIs themselves, which conveys a "place" occupied within a system of URIs. A navigation language like FOXpath, which lets us take advantage of URI structures in an unprecedented way - in the same way as XPath lets us take advantage of XML structure - might encourage a broader interest in a potential hitherto by and large ignored.

Bibliography

[1] BaseX: XML database and XQuery processor. http://basex.org

[2] EXPath Community Group. Homepage. https://www.w3.org/community/expath/

[2] Rennau, Hans-Jürgen. "FOXpath - an expression language for selecting files and folders." Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016. In Proceedings of Balisage: The Markup Conference 2016. Balisage Series on Markup Technologies, vol. 17 (2016). doi: 10.4242/BalisageVol17.Rennau01. http://www.balisage.net/Proceedings/vol17/html/Rennau01/BalisageVol17-Rennau01.html.

[3] github - how people build software. Homepage.https://github.com/

[4] Github Developer - APIhttps://developer.github.com/v3/

[5] Apache Jena - A free and open source Java framework for building Semantic Web and Linked Data applications. https://jena.apache.org/

[6] Robie, Jonathan et al, eds. XML Path Language (XPath) 3.0 W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-30/

[7] Robie, Jonathan et al, eds. XQuery 3.0: An XML Query Language. W3C Recommendation 8 April 2014. http://www.w3.org/TR/xquery-30/

[8] Walsh, Norman et al, eds. XQuery and XPath Data Model (3.0). W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-datamodel/