Task Abstraction for XPath Derived Languages

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Adam Retter
Evolved Binary
<adam@evolvedbinary.com>

Abstract

XPDLs (XPath Derived Languages) such as XQuery and XSLT have been pushed beyond the envisaged scope of their designers. Perversions such as processing Binary Streams, File System Navigation, and Asynchronous Browser DOM Mutation have all been witnessed.

Many of these novel applications of XPDLs intentionally incorporate non-sequential and/or concurrent evaluation and embrace side effects to achieve their purpose.

To arrive at a solution for safely managing side effects and concurrent execution, this paper first surveys both the available XPDL vendor extensions and approaches offered in non-XPDLs, and then describes EXPath Tasks, a novel solution derived for the safe evaluation of side effects in XPDLs which respects both sequential and concurrent execution.

1. Introduction

XPath 1.0 was originally designed to "provide a common syntax and semantics for functionality shared between XSL Transformations and XPointer" [1], and XPath 2.0 pushed the abstraction further by declaring "XPath is designed to be embedded in a host language such as XSL Transformations ... or XQuery" [2]. For XML processing, XPath has enjoyed an arguably unparalleled level of language adoption through reuse, forming the basis of XPointer, XSLT, XQuery, XForms, XProc, Schematron, JSONiq, and others. XPath has also had a wide influence outside of XML, with concepts and syntax being reused in other languages like AQL (Arrango Query Language), Cypher, JSONPath, and OData (Open Data Protocol) amongst others.

As functional languages, XPDLs such as XQuery were designed to avoid strict or ordered evaluation [21], thus leaving them open to optimisations which may exploit concurrency or parallelism. XPDLs are thus good candidates for event driven and task based concurrent and/or parallel processing. Since 2001, when the first non-embedded multi-core processor the IBM Power 4 [11] was introduced, CPU manufacturers have followed the trend of offering improved performance through greater numbers of parallel hardware threads as opposed to increased clock speeds. Unfortunately, exploiting the performance of additional hardware threads puts an additional burden on developers by requiring the use of low-level complex concurrent programming techniques [12]. Such low-level concurrent programming is often error-prone [13], so it is desirable to employ higher level abstractions such as event driven architectures [14], or task based computation with Futures [16] and Promises [18]. This paper advances the use of XPDLs in this context.

Indeed, the formal semantics for XPath state that "[XPath/XQuery] is a functional language" [4]. From this we can infer that strict XPDLs must therefore also be functional languages; this inference is strengthened by XQuery and XSLT which are both functional languages. By placing restrictions on expression formulation, composition, and evaluation, functional programming languages can enable advantageous classes of verification and optimisation when compared to imperative languages.

One such restriction enforced by functional languages is the elimination of *side effects*. A side effect is defined as a function or expression modifying some state which is external to its local environment, this includes:

1. Modifying either: a global variable, static local variable, or variable passed by reference.
2. Performing I/O.
3. Calling other side effect functions.

XPath and the XPDLs as defined by the W3C specifications address these concerns and prevent side effects by enforcing that:

1. Global variables and static local variables are immutable, and that variables are always passed by value and not reference.
2. I/O is frozen before evaluation, only documents known to the immutable static context may be read, whilst the only output facility is the XDM result of the program.
3. There are no side-effecting functions[1].

In reality though many XPDL implementations offer additional vendor-specific "*extensions*" which compromise functional integrity to permit side effects so that

[1]XPath 3.0 defines only one absolute non-deterministic function `fn:error`, and several other functions (`fn:analyze-string`, `fn:parse-xml`, `fn:parse-xml-fragment`, `fn:json-to-xml`, and `fn:transform`) which could be non-deterministic depending on implementation choices. We devalue the significance of `fn:error`'s side effect by tendering that, it could equally have been specified as a language expression for raising exceptions as opposed to a function.

I/O can be more easily achieved by the developer. Of concern for this paper is the ability to utilize XPDLs for complex I/O requiring side effects without compromising functional integrity or correctness of the application.

The key contributions of this paper are:

1. A survey of XPDL vendor implementations, detailing both how they manage side effects and any proprietary extensions they offer for concurrent execution. See Section 2.
2. A survey of currently popular mechanisms for concurrent programming in non-XPDLs, their ability to manage side effects, and their potential for XPDLs. See Section 3
3. EXPath Tasks, a module of XPath functions defined for managing computational side effects and enabling concurrent and asynchronous programming. To demonstrate the applicability of EXPath Tasks, we offer experimental reference implementations of this module in XQuery, XSLT, Java (for use from XQuery in eXist-db), and JavaScript (for use from XSLT in Saxon-JS). See Section 4.

We next briefly examine the original vision for XPath, XQuery, and XSLT, with particular concern for how these languages should be evaluated by processors. We then examine how the use of these languages has evolved over time and some of the unexpected and novel ways in which they have been used.

1.1. The vision of XPDLs

The design requirements of XPath 2.0 [3] mostly focused on that of exploiting the XDM (XQuery and XPath Data Model) and interoperability. As a language designed to describe the processing abstractions of various host languages, it did not need to state how the evaluation of such abstractions should take place, although we find that it was not without sympathy for implementations, as one of the stated Goals was: "Enable improved processor efficiency"; unfortunately, we found little explicit public information on how or if that goal was met.

Examining the XQuery 1.0 requirements [6] we find a similar focus upon the XDM, where querying different types of XML documents, and non-XML data sources is possible, provided that both can present their data in an XDM form. However, the XQuery 1.0 specification makes an explicit statement about evaluation: "an implementation is free to use any strategy or algorithm whose result conforms to the specifications in this document", thus giving implementations a great deal of freedom in how the query should be evaluated.

One of the requirements of XSLT 2.0 is labelled "2.11 Could Improve Efficiency of Transformations on Large Documents" [5]. It describes both the situation where the tree representation of source documents may exceed memory requirements, and a desire to still be able to process such large documents. It uses non-prescriptive language to suggest two possible solutions: 1) a subset of the language which would not require random access to the source tree, we could likely recognise XSLT 3.0 Streaming as the implementation of that solution, and 2) splitting a tree into sub-trees, performing a transformation on each sub-tree, and then copying the results to the final result tree. Whilst XSLT 2.0 does not state how an implementation should be achieved, many would likely recognise that (2) is an *embarrassingly parallel* problem that would likely benefit from a MapReduce [19] like approach.

An academic example of exploiting the implicit parallelisation opportunites of XPDLs is PAXQuery, which compiles a subset of XQuery down into MapReduce jobs which can execute in a highly-parallel manner over a cluster of Hadoop nodes [24]. To the best of our knowledge, Saxon is the only commercial XPDL processor which attempts implicit parallelisation. However, Michael Kay reports that within the XSLT processor it can be difficult to determine when implicitly parallelising operations will reduce processing time [20]. Saxon therefore also offers vendor extensions which allow an XSLT developer with a holistic view of both the XSLT and the data it must process, to explicitly annotate certain XSLT instructions as parallelisable.

1.2. Novel applications of XPDLs

XPDLs have been used in many novel situations for which they were never envisaged, many of which utilise non-standardised extensions for I/O side effects and concurrent processing to achieve their goals.

1.2.1. XPDLs as Web Languages

XPDLs, in particular XQuery, have been adopted with considerable success as server-side scripting languages for the creation of dynamic web pages and web APIs. A web page is by definition a document, and since an HTML document is representable as an XML document, XPDLs' ability to build and transform such documents from constituent parts has contributed to their uptake. Implementations such as BaseX, eXist-db, and MarkLogic all provide HTTP Servers which execute XQuery in response to HTTP requests. Whilst a single XQuery may be executed concurrently by many thousands of users in response to incoming HTTP requests, stateful information often needs to be persisted and shared on the server. This could be in response to either a user logging into a secure website, at which point the server must establish a session for the user and memorize the users identity; or multiple web users communicating through the server, for example, updating stock inventory for a shopping basket or social messaging. Regardless, such operations require the XPDL to make side-effecting changes to the state of the server or related systems.

XSLT's main strength as a transformation language for XML is equally applicable to constructing or styling HTML web pages. Web browsers offer limited XSLT 1.0 facilities, which can be either applied to XML documents which include an appropriate Processing Instruction, or invoked from JavaScript. The XSLT process offered by the web browser vendors is a black-box transformation, whereby an XSLT stylesheet is applied to an XML input document, which produces output. This XSLT process is completely isolated and has no knowledge of the environment from which it is called; it can not read or write directly to or from the web page displayed by the browser. In contrast, in recent years Saxonica has provided JavaScript based processors which run directly within the web browser, removing the isolation and allowing access to the web page state and events via XSLT extensions. First with Saxon-CE, a ported version of the XSLT 2.0 Saxon Java processor, and then with Saxon-JS, a clean implementation of XSLT 3.0 in JavaScript. The XSLT extensions designed for use with these processors make use of asynchronous processing (as demanded by JavaScript) and side effects to read and write the DOM model of the web page.

Similar to Saxon-CE, although now unmaintained another notable example is XQiB. XQiB implements an XQuery 1.0 processor in JavaScript which runs in the web browser and provides a number of XQuery extension functions which cause side effects by writing directly to the HTML DOM and CSS [25].

1.2.2. Binary Processing with XPDLs

The generation of various binary formats using XPDLs has also been demonstrated. One such example is Philip Fennel's generation of TIFF format images, which uses a Reyes pipeline written in XSLT [26]. One of Fennel's conclusions with regard to execution was that “Certainly it is not fast and it is not very efficient either”. It is not hard to imagine that if concurrent processing was applied to each stage of the pipeline, so that stages were processed in parallel, then execution time might be significantly reduced.

Two XPath function extension modules produced by the EXPath project, the Binary [27] and File Module [28] specifications, allow the user to both read and write files and manipulate binary data at the byte level from within XPDLs. In particular, the File Module, which provides I/O functions, states that some functions are labelled as *non-deterministic*; this specification lacks the detail required to determine if implementations are forced to produce side effects when the functions are evaluated, or whether they are allowed to operate on a static context and only apply I/O updates after execution. The authors of this paper believe that it would be beneficial to have a more formal model within that specification, possibly one which allows implementers flexibility to determine the scope of side effects.

1.3. Motivation

To enable side effects in a web application running in Saxon-JS, the IXSL (Interactive XSLT) extensions (instructions, functions and modes) are provided (as previously developed for Saxon-CE, with some further additions and improvements). These IXSL extensions allow rich interactive client-side applications to be written directly in XSLT.

Saxon-CE used a Pending Update List (PUL) to make all HTML page DOM updates (side effects) at the end of a transform (e.g. setting attributes on HTML page nodes using `ixsl:set-attribute`; and adding content to the HTML page using `xsl:result-document`.) Currently Saxon-JS does not use a PUL, instead these side-effecting changes are allowed to execute immediately as the instructions are evaluated, and it is up to the developer of a Saxon-JS application to ensure that adverse affects are avoided. Since inception, the intention has been to eventually provide better implicit handling. Should the use of PULs be reinstated, or is there an alternative solution?

Meanwhile, use of asynchronous (concurrent) processing is essential for user-friendly modern web applications. Whenever the client-side needs to interact with the server-side, to retrieve resources, or make other HTTP requests, this should be done asynchronously. The application may provide a "processing, please wait" message to the user, but it should not just stop due to blocking.

The `ixsl:schedule-action` instruction allows the developer to make use of concurrent threads, and in particular allows for asynchronous processing. In Saxon-JS, different attributes are defined to cater for specific cases where there is a known need. The `document` attribute is used to initiate asynchronous document fetches; the `http-request` attribute is used for making general asynchronous HTTP requests; and the `wait` attribute was designed to force a delay (e.g. to enable animation), but actually simply provides a way to start any concurrent process. Effectively this provides a mechanism for forking, but there is no offical joining. Are there cases that require a join? Are there other operations which a developer could want to make asynchronously? Rather than building IXSL extensions for each operation, we would prefer to realise a general mechanism for asynchronous processing in XPDLs and by extension XSLT. Continually updating the syntax and implementation of `ixsl:schedule-action`, each time a new requirement arises (e.g. how to allow HTTP requests to be aborted), is not ideal. In particular, the IXSL HTTP request facility was based on the first EXPath HTTP Client Module, recent work on a second version [23] of that module could be advantageous for us. However, by itself it neither prescribes synchronous or asynchronous operation. So, how could we implement in a manner which is both asynchronous and more abstract, requiring few, if any, changes to add additional modules in future?

1.4. Our Requirements

Applications that cannot perform I/O and/or launch parallel processes are unusual. Both I/O and starting parallel processes are side effects, and as discussed, explicitly forbidden within XPDLs, although often permitted by vendors at the cost of imperative interpretation and lost optimisation opportunities.

We aim to break the trade-off between program correctness and deoptimisation in XPDLs. We require a mechanism that satisfies the following requirements:

- A mechanism for formulating processes which manage side effects, yet at the same time remains within the pure functional approach dictated by the XPDL formal semantics.
- Permits some form of parallel or concurrent operation, that is implementable on systems that offer either preemptive or cooperative multitasking.
- Allows parallelisation to be explicitly described, but that should not limit the opportunities for implicit parallelisation.
- Any parallel operation explicitly initiated by the developer, should be cancellable.
- Composability: it should be possible to explicitly compose many side-effecting processes together in a manner that is both pure and ensures correct order of execution.

Regardless of the mechanism, we require that it should be as widely applicable as possible, therefore it should be either:

- Formulated strictly in terms of XPath constructs so that it be reused by any XPDL.

  Ideally, rather than developing a superset of the XPath grammar, a module of XPath extension functions should be defined. The module approach has been successfully demonstrated by the EXPath project, and would likely lower the barrier to adoption.
- A clearly defined abstract processing model which can have multiple syntactical expressions.

  Such a model could for example provide one function-based syntax for XQuery, and another instruction-based syntax for XSLT.

2. Current Approaches by Implementers

This survey provides a brief review of the offerings of the most visible XQuery and XSLT implementations for both concurrent and/or asynchronous execution, and how they manage side effects.

2.1. BaseX

For concurrent processing from within XQuery, BaseX provides two mechanisms: a Jobs XQuery extension module [8], and an XQuery extension function called `xquery:fork-join`. The latter is actually an adoption of xq-promises's `promise:fork-join` function, which we cover in detail in Section 2.5. The former, the Jobs Module, allows an XQuery to start another XQuery by calling either of two XPath functions `jobs:invoke` or `jobs:eval`. Options can be supplied to both of these functions, which instead of executing the query immediately, schedule it for later execution. Whilst deferred scheduled queries are possibly executed concurrently we will not consider them further here as our focus is concurrent processing for the purposes of completing an immediate task. BaseX describes these functions as *asynchronous*, and whilst technically true, unlike other asynchronous programming models the caller neither provides a callback function nor receives a promise, and instead has to either poll or wait in the main query for the result. We believe these functions could more aptly be described as *non-blocking*.

Asynchronously starting another XQuery in BaseX returns an identifier which can be used to either stop the asynchronously executing query, retrieve its result (if it has completed), or to wait until it has completed. The lifetime of the asynchronously executing query is not dependent on the initiating query, and may continue executing after the main query has completed. In many ways this is very similar to a Future (see Section 3.5).

BaseX implements XQuery Update [7] which allows updates to XML nodes to be described from within XQuery via additional update statement syntax. XQuery Update makes use of a PUL (Pending Update List) which holds a set of Update Primitives. These Update Primitives describe changes that will be made, but have not yet been applied. These changes are not visible to the executing query, the PUL is output alongside the XDM when the query completes. This is not entirely dissimilar to how Haskell eventually evaluates an IO monad (see Section 3.4). To further facilitate additional common tasks required in a document database without conflicting with XQuery Update or resorting to side effects within an executing query, BaseX also provides many vendor specific Update Primitives in addition to those of XQuery Update. These include primitives for database operations to replace, rename and delete documents; manage users; and backup and restore databases [29]. The use of an XQuery Update PUL avoids side effects for updates, as it only describes what will happen at evaluation time, leaving the actual updates to be applied at execution time. Ultimately BaseX applies the PUL to modify the state of its database after the query completes and the transaction is committed, thus making the updates visible to subsequent transactions.

Regardless of its support for PULs, BaseX does not quite manage to entirely avoid side effects during the execution of some queries. BaseX offers a number of

XQuery extension functions which are known to cause side effects, including for example, those of the EXPath HTTP and File Modules. Internally such side-effecting functions are annotated as *nondeterministic*, and will be treated differently by BaseX's query compiler. By skipping a range of otherwise possible query optimisations, BaseX ensures that the execution order of the functions within a query is as a user would expect even when these nondeterministic functions are present. In the presence of nondeterminism, optimisations that are skipped include: pre-evaluation, reordering of let clauses, variable inlining, and disposal of expressions that yield an empty sequence.

2.2. eXist-db

eXist-db does not present a cohesive solution for concurrent processing from within XQuery. Until recently, eXist-db had a non-blocking XPath extension function named `util:eval-async` [9] which could start another XQuery asynchronously. Like BaseX it returned an identifier for the executing query and did not accept a callback function or provide a promise. Unlike BaseX however, there were no additional functions to control the asynchronously executing query or obtain its result, rather the asynchronously executing query would run to completion and its result would be discarded, although it may have updated the database via side effects. This facility proved not to be particularly practical and has since been removed. Similarly to BaseX, eXist-db provides a Scheduler XQuery extension module [10] for scheduling the future (or immediate) execution of jobs written in XQuery. Unfortunately even if an XQuery is scheduled for immediate execution, there is no mechanism for obtaining the result of its execution from the initiating XQuery.

eXist-db makes no attempts to avoid side effects during processing, and instead offers many extension functions and a syntax for updating nodes that cause side effects by immediately modifying external state and making the modifications visible. eXist-db also relaxes the XPath deterministic constraint upon Available Documents, and Available Collections, allowing a query to both modify which documents and collections are available (a side effect), and to see changes made by concurrently executing queries.

eXist-db is able to suffer side effects, through making several compromises:

- eXist-db offers the lowest transaction isolation level when executing XQuery - Read Uncommitted.

  eXist-db makes XQuery users somewhat aware of this, and provides XPath extension functions which enable them to lock documents and collections on demand if they require a stronger isolation level.

- eXist-db executes XQuery sequentially as though it was a procedural program.

Whilst some query rewriting is employed to improve performance, eXist-db cannot exploit many of the more advanced optimisations available to functional language compilers: any reordering of the XQuery program's execution path could cause the program to return incorrect results, due to side effects being applied in an order that the XQuery developer had not intended.

Likewise, eXist-db cannot easily parallelise the execution of disjoint statements within an XQuery: as shared-state modified by side effects could introduce race conditions in the XQuery developer's application.

2.3. MarkLogic

MarkLogic provides an XPath extension function named `xdmp:spawn`, which allows another XQuery to be started asynchronously from the calling query. This is done by placing it on the task queue of the MarkLogic task server, and this query may be executed concurrently if the task server has the available resources. The function is *non-blocking*, and for our interests has two modes of operation controlled by an option called `result`. When the *result* option is set to *false*, the calling query has no reference to the queued query, and like eXist-db it can neither retrieve its result, enquire about its status, or abort its execution. When the *result* option is set to *true*, the `xdmp:spawn` function returns what MarkLogic describes as a “value future for the result of the spawned task”. This “value future” is quite unusual, and certainly a vendor extension with no corresponding type in XDM. Essentially, after calling `xdmp:spawn` with the *return* option set to *true*, the calling query continues executing until it tries to access the value of the variable bound to the result of the `xdmp:spawn`, at which point if the *spawned* query has completed executing, the result is available, however if it has not completed then the main query thread blocks and waits for the *spawned* query to complete and provide the result [30]. Similarly to BaseX and eXist-db, MarkLogic also provides mechanisms for the scheduling of XQuery execution through its offline batch processing framework called CPF (Content Processing Framework) [31], and a set of XPath extension functions such as `admin:group-add-scheduled-task` [32].

MarkLogic's *value future* is intriguing in its nature, albeit proprietary. The concept of Futures appear in several programming languages, but unlike other languages (e.g., Java or C++11), MarkLogic's implementation provides no explicit call to *get* the value of the future (possibly with a timeout), instead the *wait* and/or *get* happen as one implicitly when accessing the value through its variable binding.

MarkLogic clearly documents where it allows side effects from within XQuery. There are two distinct types of side effects within MarkLogic, state changes that happen within the scope of the XQuery itself, and those state-changes which are external to the XQuery. For use within the scope of an XQuery, MarkLogic provides an XPath extension function `xdmp:set`, which explicitly states that it uses “changes to the state (side effects)” [33] to modify the value of a previously declared variable, thus violating the formal semantics of XPath [4]. For modifying state external to an XQuery, MarkLogic provides a series of XPath extension functions for updating nodes and managing documents within the database. Similarly to BaseX, these extension functions do not cause side effects by immediate application, and are invisible to both the executing query and concurrently executing queries [34]. Unlike BaseX, MarkLogic does not implement the XQuery Update specification, but similarly it utilizes a PUL, likewise leading to a process whereby the updates are applied to the database after the query completes and the transaction is committed, thus making the updates visible to subsequent transactions.

Whilst MarkLogic utilizes both a well defined transaction isolation model and deferred updates to mostly avoid side effects within an executing XQuery, we suspect that the use of `xdmp:set` likely places some limitations on possible query optimisations that could be performed.

We have focused on MarkLogic's XQuery implementation, but it is worth noting that MarkLogic also implements XSLT 2.0. All of MarkLogic's XPath extension functions (e.g., `xdmp:set` and `xdmp:insert-*`) are also available from its XSLT processor, and are subject to the same transactional mechanisms as the XQuery processor; therefore our findings are equally applicable to running either XQuery or XSLT on MarkLogic.

2.4. Saxon

Saxon-EE utilises parallel processing in certain specific instances [20]. By default the parsing of input files for the `fn:collection` function is multithreaded, as is the processing of `xsl:result-document` instructions. Note that the outputs produced by multiple `xsl:result-document` instructions are quite independent and never need to be merged; so while this does allow parallel execution of user code and requires careful implementation of features such as try/catch and lazy evaluation, the fact that there is a "fork" with no "join" simplifies things a lot. Furthermore, multi-threading of `xsl:for-each` instructions using a MapReduce approach can be enabled by the user, by setting the `saxon:threads` extension attribute to specify the number of threads to be used.

Saxon-EE allows use of a number of extension functions with side effects, including those in the EXPath File and Binary modules. Similar to the BaseX handling, the Saxon compiler recognises such expressions as causing side effects, and takes a pragmatic approach in attempting to avoid aggressive optimisations which could otherwise disrupt the execution order. Usually instructions in an XSLT sequence constructor will be executed sequentially in the order written, but deviation can be caused by the compiler through lazy evaluation or loop lifting; and this is where problems can arise when side effects are involved. Such optimisations can cause the side effect to happen the wrong number of times (never, or too often), or at the wrong time. It is relatively straightforward to prevent such optimisations for static calls to side-effecting functions, but cannot always be guaranteed for more nested calls, as "side-effecting" is not necessarily recognised as a transitive property. For instance, a function or template which includes a call to a side-effecting function may not itself be recognised as side-effecting. So it is always recommended that side-effecting XPath expressions are "used with care". One mechanism which gives the XSLT author better control when using side-effecting expressions, is the recently added extension instruction `saxon:do`. It is similar to the `xsl:sequence` instruction, but is designed specifically for use when invoking XPath expressions with side effects. In contrast to `xsl:sequence`, when using `saxon:do` any result is always discarded, and the processor ensures that instructions in the sequence constructor are always evaluated sequentially in the order written, avoiding any reordering from optimisations.

As previously mentioned, for use with the Saxon-JS runtime XSLT processor, a number of Interactive XSL extension instructions and functions are available. To enable non-blocking (asynchronous) HTTP requests and document fetching, the `ixsl:schedule-action` instruction is provided. Attributes on the instruction are used to specify an HTTP request, or document URI, and the associated HTTP request is then executed in a new concurrent thread. The callback, for when an HTTP response is returned or the document is fetched (or an HTTP error occurs), is specified using the single permitted `xsl:call-template` child of the `ixsl:schedule-action` instruction. When the `document` attribute has been used, the called template can then access the document(s) using the `fn:doc` or `fn:doc-available` functions; the document(s) will be found in a local cache and will not involve another request to the server. When using the `http-request` attribute, the HTTP response is supplied as the context item to the called template, in the form of an XDM map. Alternatively, `ixsl:schedule-action` can simply be used to start concurrent processing for any action, by using just the `wait` attribute (with a minimal delay). Note that while this provides a "fork", there is no "join", and it is up to the developer to avoid conflicts caused by side effects.

To be able to write interactive applications directly in XSLT, it is necessary to make use of side effects, for example to dynamically update nodes in the HTML page. Almost all of the IXSL extension instructions and functions (such as `ixsl:set-attribute` and `ixsl:set-property` which are used to set attributes on nodes and properties on JavaScript objects respectively) have (or may have) side effects. Note that Saxon-JS runs precompiled XSLT stylesheets, called SEFs (Stylesheet Export Files) generated using Saxon-EE. As described above, during compilation in Saxon-EE, such side-effecting functions and instructions are internally marked as such to prevent optimisations from disrupting the intended execution order.

2.5. xq-promise

Whilst xq-promise [35] is not an implementation of XQuery or XSLT, it is the first known non-vendor specific proposal for a module of XPath extension functions by which XPDL implementations can offer concurrent processing from within an XPDL. It is valuable to review this proposal as theoretically it could be implemented by any XPDL implementation, at present we are only aware of a single implementation for BaseX [36].

xq-promise first and foremost provides a set of XPath extension functions which were inspired by jQuery's Deferred Object utility, it claims to implement the "promise pattern" (see Section 3.5), and focuses on the concept of deferring execution. In its simplest form, the `promise:defer` function takes two parameters: a function of variable arity, and a sequence of arguments of the same arity as the function. Calling `promise:defer` returns a new zero arity function called a "promise", this *promise* function encapsulates the application of the function passed as a parameter to the arguments passed as a parameter. The encapsulation provided by the promise function *defers* the execution of the encapsulated function. The promise function also serves to enable chaining further actions which are dependent on the result of executing the deferred function, such further actions are also deferred. The chaining is implemented through function composition, but is opaque to the user who is provided with the more technically accessible functions `promise:then`, `promise:done`, `promise:always`, `promise:fail`, and `promise:when`.

The functions provided by xq-promise discussed so far allow a user to describe a chain of related actions, where callback functions, for example established through `promise:then`, can be invoked when another function completes with success or failure. Considered in isolation these functions do not explicitly prescribe any asynchronous or concurrent operation. To address this, xq-promise secondly provides an XPath extension function named `promise:fork-join` based on the Fork-join model of concurrency. This functions takes as a parameter a sequence of promise functions, which may then be executed concurrently. The `promise:fork-join` function is a blocking function, which is quite different from those of BaseX, eXist-db, MarkLogic, or Saxon, which are all non-blocking. Rather than scheduling a query for concurrent execution and then returning to the main query so execution can continue, when `promise:fork-join` is invoked *n* query sub-processes are *forked* from the main query which then waits for these to complete, at which point the results of the sub-processes are *joined* together and returned as the result of the function call.

An important insight we offer is that whilst sharing some terminology with implementations in other languages (particularly JavaScript likely due to building upon jQuery's Deferred Object) the *promise* concept used in xq-promise is subtly different [61]. JavaScript Promises upon construction immediately execute the function that they are provided [38] [39], whereas an xq-promise is not executed until either `promise:fork-join` is used or the promise function is manually applied by the user. Conceptually the xq-promise promises appear to be at odds with the fork-join approach, as once a promise has been constructed, it is likely that useful computation could have been achieved in parallel to the main thread by executing the promise(s) before reaching the fork-join point. The construction of a JavaScript Promise requires an *executor* function, which takes two parameter functions, a resolve function and a reject function. The executor must then call one of these two functions to signal completion. When constructing a promise with xq-promise, completion is instead signalled by the function terminating normally, or raising an XPath error. This may appear to be just syntactical differences, but the distinction is important: the JavaScript approach allows an error value to explicitly be returned upon failure in a functional manner, the xq-promise approach relies instead on `fn:error`... which is a side effect!

On the subject of xq-promise and side effects, xq-promise constructs chains of execution where each step has an dependency on the result of the preceding step. On the surface this may appear similar to how IO Monads (see Section 3.4) compose. The composition of xq-promise through is much more limited, and whilst it ensures some order of execution, its functional semantics are likely not strong enough to ensure a total ordering of execution.

2.6. Conclusion of Implementers Survey

Our conclusion from this survey is twofold. Firstly, all surveyed implementations offer some varying proprietary mechanism for performing asynchronous computations from within a main XPDL thread of execution. A standardised approach is evidently missing from the W3C defined XPDLs, but a requirement has been demonstrated by implementations presumably meeting a technical demand of their users of XPDLs. Secondly, none of the XPDL implementations which we examined adhere strictly to the functional processing semantics required by XPath and/or the respectively implemented XPDL specification. Instead each implementation to a lesser or greater extent offers some operations which cause side effects. Most implementations appear to have taken a pragmatic approach to deliver the features that their users require, often sacrificing the advantages of a pure functional approach to offer a likely more familiar imperative programming model.

3. Solutions offered for non-XPDLs

This survey provides a brief review of several options for non-XPDLs that provide solutions for both concurrent and/or asynchronous execution, and how side effects are managed or avoided. This is not intended as an exhaustive survey, rather the options surveyed herein were subjectively chosen for their variety.

3.1. Actor Model

The Actor Model defines a universal concept, the Actor, which receives messages and undertakes computation in response to a message. Each Actor may also asynchronously send messages to other Actors. A system is typically made up of many of these Actors [40]. Actor systems are another class of embarrassingly parallel problem, as the messages sent between actors are immutable, there is no shared-mutable state to synchronize access to, and so each Actor can run concurrently.

The Actor Model by itself is not enough to clearly describe, manage, or eliminate side-effectful computation, however by nature of its message passing approach it does eliminate the side effects of modifying the shared-state for communication between concurrent threads of execution which is often found in non-actor systems. Through encapsulation, actors may also help to reason about programs with side effects. Systems utilising actors are often built in such a manner that each task specific side-effectful computation is isolated and encapsulated within a single Actor. For example, within an actor system there may only be a single Actor which handles a particular file I/O, then since each Actor likely runs as a separate process, the file I/O has been isolated away from other computation.

The Erlang programming language is possibly the most well known Actor Model like implementation, wherein Actors are known as processes [41]. Erlang itself makes no additional efforts to manage side effects, and additional synchronization primitives are often employed. Within the JVM (Java Virtual Machine) ecosystem, the Akka framework is available for both Java and Scala programming languages [42]. Java as a non-functional language makes no attempts at limiting side effects. Meanwhile, whilst Scala is often discussed as a functional language and does provide many functional programming constructs, it is likely more a general purpose language, as mutability and side effects are not restricted, and it is quite possible to write imperative Scala code. Actor systems are also available for many other programming languages [43], although they do not seem to have gained the same respective popularity as Erlang or Akka.

3.2. Async/Await

The Async/Await concept was first introduced in C#, inspired by F#'s *async workflows* [44], which was in turn inspired by Haskell's Async Monad [45] [46] (see Section 3.4). Async/Await provides syntax extensions to a programming language in the form of the `async` and `await` keywords. Async/Await allows a developer to write a program using a familiar synchronous like syntax but easily achieve asynchronous operation of parts of the program.

Async/Await adds no further processing semantics for concurrency or managing side effects over that of Promises (see Section 3.5), which are often used to implement Async/Await. Async/Await may be thought of as syntactic sugar for utilising a Promise based implementation, and has recently become very popular with JavaScript developers [47] [48].

3.3. Coroutines

Coroutines are a concept for cooperative multitasking between two (or more) processes within a program. One process within an application, Process A, may *explicitly* yield control to another process, Process B. When control is transferred, the state of Process A is saved, the current state of Process B is restored (or a new state created if there is no previous state), and Process B continues until it *explicitly* yields control back to Process A or elsewhere [49].

Like Actors, the impact of side effects of impure functions can be somewhat isolated within a system by encapsulating them in distinct coroutines. Otherwise Coroutines provide no additional facilities for directly managing side effects, and global state is often shared between them. Unlike Actors, Coroutines are often executed concurrently by means of explicitly yielding control. Without additional control structures, coroutines typically operate on a single-thread, one exception is Kotlin's Coroutines which can be structured to execute concurrently across threads [52].

Some implementations of Coroutines, such as those present in Unity [50], or JavaScript [51], attempt to bring a familiar synchronous programming style to the developer. These implementations typically have a coroutine *yield* multiple results to the caller, as opposed to yielding control. This masks the cooperative multitasking aspect from the developer and presents the return value of a coroutine as an iterable collection of results.

3.4. IO Monads

Haskell is a statically typed, non-strict, pure functional programming language. The *pure* aspect means that every function in Haskell must be *pure*, that is to say akin to a mathematical function in the sense that mathematical functions cannot produce side effects. Even though Haskell prohibits side effects by design, it still enables developers to perform I/O and compute concurrently. This seemingly impassable juxtaposition of academic purism and real-world engineering need is made possible by its IO Monad [54]. Haskell trialled several other approaches in the past, including streams and continuations, before the IO Monad won out as it facilitated a more natural imperative programming style [55].

In Haskell, any function that performs I/O must return an IO type which is monadic. This IO type represents an IO action which has not yet happened. For example if you have a function that reads a string from a file, that function does not directly return a `String`, instead it returns an `IO String`. This is not the result of reading a line from the file, instead it can be thought of as an action that when *executed* will read a line from the file and return a `String`. These IO actions describe the I/O that you wish to perform, but critically defer its execution. The `IO` actions adhere to *monad laws* which allow them to be composed together. For example given two IO actions, one that reads a file and one that writes a file, they could be composed together into a single IO action which first reads a file and then writes a file, e.g. a copy file IO action.

Importantly, the formal definition for an `IO` type is effectively `IO a = World -> (a, World)`. That is to say that an IO is a state transformation function that takes as input the current state of the world, and produces as the result both a value and a new state of the new world. The `World` is a purely Abstract Data Type, that the Haskell programmer cannot create. The important thing to note here is that the `World` is threaded through the IO function. When multiple IO actions are composed together using monadic application, such as *bind*, the `World` output from a preceding function will be fed to the input of the succeeding function. In this manner the `World` will be threaded through the entire chain of IO actions.

A Haskell program begins by executing a function named `main` that must return an IO, it is typed as `mainIO :: IO ()`. Haskell knows how to execute the IO type function that the main function returns. Naively one can think of this as Haskell's runtime creating the `World` and then calling our IO with it as an argument to execute our code; in reality the Haskell compiler optimises out the `World` during compilation whilst still ensuring the correct execution order. (We may remark that an IO action is similar to a PUL's Update Primitive, and the fact that `main` returns an IO is not dissimilar to an XQuery Update returning both XDM and a PUL.)

By using IO Monads which defer rather than perform I/O, all Haskell functions are pure, and so a Haskell program at evaluation time exhibits no side effects whatsoever, instead finally evaluating to an `IO ()`, i.e. a state transformation function upon the world. As the developer has used monadic composition of their IO actions, this has implicitly threaded the `World` between them, in the order the developer would expect (i.e. in the order of the composition), therefore the state transformation also ensures that the functions are executed in the expected/correct order. At execution time, the machine code representation of the Haskell program is run by a CPU which is side-effecting in nature, and the IO action's side effects are unleashed.

> *It is possible to encapsulate stateful computations so that they appear to the rest of the program as pure (stateless) functions which are guaranteed by the type system to have no interactions whatever with other computations, whether stateful or otherwise (except via the values of arguments and results, of course).*
>
> *—from "State in Haskell", by John Launchbury and Simon Peyton Jones*

Haskell provides further functions for concurrency, but critically these also return IO actions. One such example is `forkIO` with the signature `forkIO :: IO () ->`

`IO ThreadId` [56]. The purpose of forkIO is to execute an IO in another thread, so it takes an IO as an argument, and returns an IO. The important thing to remember here, is that calling the forkIO function does not create a new thread and execute an IO, rather it returns an IO action which describes and defers such behaviour. Later when this IO action is finally executed at run-time, the thread will be created at the appropriate point within the running program. There are also a number of other higher-level abstractions for concurrency in Haskell, such as Async [46], and whilst such abstractions *may* introduce additional monads, they ultimately all operate with IO to defer any non-pure computation. One final point on the IO Monad, is to mention that concurrently executing I/O actions, may at runtime produce side effects that conflict with each other. The IO Monad is only strong enough to ensure correct operation within a single thread of execution, its protections do not cross thread-boundaries. To guard against problems with concurrent modifications additional synchronisation is required. Haskell provides additional libraries of such functions and types for working with synchronization primitives, many of which themselves produce IO actions!

Monads are by no means limited to Haskell, and can likely be used in any language which supports higher-order functions. The preoccupation with Haskell is centred around how it uses Monads to ensure a pure language in the face needing to perform I/O. Several libraries exist which attempt to bring the IO Monad concept to other programming languages, this seems to have been most visible within the Scala ecosystem, where there are now at least five differing established libraries [57]. Whilst all of these efforts are admirable and bring new mechanisms for managing side effects, they all have one weakness which Haskell does not: in Haskell one is forced to ensure that the entire program is pure, because the main function must return an IO. The runtimes of other languages are not structured in this way, and so these IO Monad libraries are forced to rely on workarounds to evaluate the IO. These rely on the user structuring their program around the concept of an IO, and only evaluating that IO as the last operation in their program. For example Monix Task [58], where the user must eventually call `runUnsafeSync` to evaluate the IO, describes the situation as thus:

> *In general prefer to ... structure your logic around asynchronous actions in a non-blocking way. But in case you're blocking only once, in* `main`*, at the "edge of the world" so to speak, then it's OK.*
>
> —*Alexandru Nedelcu*

3.5. Promises and Futures

There may be some confusion over the differences between the computer science terms *Promise, Future*, or even *Eventuals*. However, these terms are academically synonymous, as perhaps best explained by Baker and Hewitt, the fathers of the term *Future* [16]:

the mechanism of futures, which are roughly Algol-60 "thunks" which have their own evaluator process ("thinks"?). (Friedman and Wise [18] call futures "promises", while Hibbard [17] calls them "eventuals".)

—Henry G. Baker Jr. and Carl Hewitt

The confusion likely comes from implementations that offer both Future and Promise abstractions to developers looking for safer concurrency facilities, yet use differing terminology and provide vastly different APIs. Two examples of extreme variation of terminology, are the Scala and Clojure programming languages, which each define Future and Promise as distinct classes. The Scala/Clojure Future class is much more like the computer science definition of Future/Promise which models computation; whereas the Scala/Clojure Promise class serves a very different purpose, primarily as a memorized data provider for completing a Future class. We are strictly interested in the computer science definition of Promise and Future, and herein will refer to them singly as Promise.

A Promise represents a value which may not yet have been computed. Typically when creating a Promise a computation is immediately started asynchronously and returns a Promise. In implementation terms, a Promise is a reference which will likely take the form of an object, function, or integer. At some point in the future when the asynchronous computation completes, the Promise is fulfilled with the result of the computation which may be either a value or an error. Promises provide developers with an abstraction for concurrent programming, but whether that is executed via cooperative or preemptive multi-tasking is defined by the implementation. Promises by themselves provide no mechanism for avoiding side effects as they are likely eagerly evaluated, with multiple promises being unordered with respect to execution.

Some implementations, for example those based on Promise/A+ like JavaScript, allow you to functionally compose Promises together [53]. This functional composition can allow you to chain together side-effecting functions which are encapsulated within Promises, thus giving an explicit execution order, in a manner not dissimilar to Haskell's IO Monad (see Section 3.4). Unlike Haskell's IO Monad however, this doesn't suddenly mean that your application is pure: remember that JavaScript Promises are eagerly evaluated. It does though offer a judicious JavaScript developer some measure to ensure the correct execution order of her impure asynchronous code.

3.6. Reactive Streams

Reactive Streams enable the composition of a stream of computation, where the Publisher, Subscriber, or a Processor in the stream (which act as both Subscriber and Publisher), may operate asynchronously [59]. A key characteristic of Reactive Streams is that of *back-pressure*, a form of flow control which can prevent slower Subscribers from being overwhelmed by faster asynchronous Producers. This built-in back-pressure facility appears to be unique to Reactive Streams, and would otherwise have to be manually built by a developer atop other concurrency mechanisms.

The Reactive Streams initiative itself just defines a set of interfaces and principles for Reactive Stream implementations, it is up to the implementations to provide mechanisms for controlling concurrent and parallel processing of streaming values. Typically implementations provide mechanisms for parallelising Processors within a stream, or splitting a stream into many asynchronously executing streams which are later resolved back to the main stream.

Reactive Streams offers little explicitly to help with side effects, however if we consider that a data flow within a non-concurrent stream is always downwards, then streams do provide an almost Monadic-like mechanism for composing processing steps where the order of execution becomes explicit. Likewise, if one was to ensure that the data that is passed from one step to another is immutable, then when there are concurrent or asynchronous Subscribers, there can be no data-driver side effects between them as the data provided by the publisher was immutable, meaning that any changes to the data by a subscriber are isolated to a localised copy of the data.

Examples of Reactive Streams implementations that support concurrent and parallel processing at this time include: RxJava, Akka Streams, Monix, Most.js, and Reactive Streams .NET#

3.7. Conclusion of non-XPDL Solutions Survey

Our survey shows several different options for concurrent/parallel programming. It is possible to build the same application using any of these options, but each offers a different approach and syntax for isolating and managing concurrently executing processes. As well as the underlying computer science principles of each option, the libraries or languages that implement these options can vary between Cooperative Multitasking and Preemptive Multitasking. Coroutines, Async/Await, and Promises are particularly well suited to Cooperative Multitasking systems due to their explicit demarcation of computation boundaries, which can be used to yield the CPU to another process. Likely this is why these options have been adopted in the JavaScript community, where JavaScript Virtual Machines are frequently designed as cooperatively multitasking systems utilising an event loop [60].

We find that the IO Monad is the only surveyed option that is specifically designed to manage computational side effects in a functional manner. This is likely due to the fact that the IO Monad approach was explicitly developed for use in a non-strict purely functional language, i.e. Haskell, whereas all of the other approaches are more generalised, and whilst not explicitly limited to imperative languages are often found in that domain.

Of all the approaches surveyed, to the best of our knowledge, only the development of a Promise-like approach has been realised for XPDLs, namely xq-promise (see Section 2.5). It seems likely that at least aspects of the IO Monad approach (such as that demonstrated by Monix), or Reactive Streams options, could be implemented by utilising XPath extension functions and a written specification of concurrent implementation behaviour, without resorting to proprietary XPath syntax extensions. Conversely, whilst an XPath function based implementation could likely be devised, both Async/Await and Coroutines would likely benefit by extending the XPath language with additional syntax.

In conclusion, we believe that an IO Monad exhibits many of the desirable properties that we set out to discover in Section 1.4. It has strong pure functional properties, strict isolation of side effects, and acts as a building block for constructing further concurrent/parallel processing. Therefore we have chosen to use this as the basis for a solution to handle side effects and sequential or concurrent processing in XPDLs.

4. EXPath Tasks

Herein we describe EXPath Tasks, a module of extension XPath functions for performing *Tasks*. These functions have been designed to allow an XPDL developer to work with both side effects and concurrency in a manner which appears imperative but is functionally pure, and therefore does not require processors to sacrifice optimisation opportunities.

The specification of the functions and their behaviour is defined in Appendix A. We have also developed four reference implementations:

XQuery	task.xq is written in pure XQuery 3.1 with no extensions. It implements all functions, however all potentially asynchronous operations are executed sychronously. The source code is available from https://github.com/adamretter/task.xq.
XSLT	task.xsl is written in pure XSLT 3.0 with no extensions. There is a lot of code overlap with task.xq, since much is actually XPath 3.1. Like task.xq, it implements all functions, however all potentially asynchronous operations are executed sychronously. The source code is available from https://github.com/saxonica/expath-task-xslt.
Java	An implementation of EXPath Tasks for XQuery in eXist-db. The source code is available from https://github.com/eXist-db/exist/tree/expath-task-module-4.x.x/extensions/expath/src/org/expath/task.
JavaScript	An implementation of EXPath Tasks for XSLT in Saxon-JS.

4.1. The Design of EXPath Tasks

From the findings of our survey on non-XPDL solutions (see Section 3), we felt that the best fit for our requirements (see Section 1.4) was that of developing a module of XPath Functions that could both ensure the correct execution ordering of side-effecting functions, and provide facilities for asynchronous programming.

We decided to adopt the principles of the IO Monad, as we have previously identified it as providing the most comprehensive approach to managing non-deterministic functions in a pure functional language. Our design was heavily influenced by both Haskell's IO [54] and Async [46] packages, and to a lesser extent by Monix's Task [58].

Our decision to develop a module of extension functions rather than grammar extensions, was influenced by a previous monadic approach for XQuery, called *XQuery!*, which utilized grammar extensions but failed to gain adoption [63].

An astute reader may raise the question of why we didn't attempt a translation of IO actions to PUL Update Primitives. The issue that we saw is that a PUL is an opaque collection, which cannot be computed over. With XQuery Update there is no mechanism for directly working with the result of a previous Update Primitive. We required a solution that was applicable to general computation, so we focused on a task based approach. Of course there is the concern that we would have also had to adopt much of the XQuery Update specification to make this work in practice. For XPDLs that are not derived from XQuery this may have been prohibitive to adoption. However, we see no reason why further work could not examine the feasibility of *lifting* a Task to an Update Primitive.

4.1.1. Abstract Data Types

Haskell's IO Monad makes use of an ADT (Abstract Data Type) to represent the *World* which it is transforming. The beauty of using an ADT here is that the Haskell programmer cannot themselves instantiate this type[2], which makes it impossible to execute IO directly. Instead the Haskell compiler is responsible for compiling the application in such a manner that the IO will be implicitly executed at runtime.

Recall that the IO type is really a state transformation function, with the signature

```
IO a = World -> (a, World)
```

To create an equivalent function for XPDLs we need some mechanism for modelling the World ADT. Unfortunately, without requiring *Schema Awareness*, the

[2]Haskell does provide an `unsafePerformIO` function which can *conjure* the world up, and execute the IO. However, such behaviour is considered bad practice in the extreme.

XDM type system is sealed. It is not possible to define new types abstract or otherwise within XPDLs.

To remain within the XPDL specifications we must therefore define the World using some non-abstract existing type. Unfortunately, this means that the developer can also instantiate the World and potentially execute the IO. We developed an initial prototype [62] where we modelled the World simply as an XDM Element named `io:realworld`, thus our XPath IO type function was defined such:

```
declare function io:IO($realworld as element(io:realworld)) as item()+
```

Note the item()+ return type: in XPath there is no tuple type so we have to use a less strict definition than we would prefer. This sequence of items will have 1+*n* items, where the head of the sequence is always the new state of the world (i.e. the XDM element named `io:realworld`), and the tail of the sequence is the result of executing the IO.

Implementations written for XPDLs in non-XPDLs could likely enforce stronger semantics by using some proprietary type outside of the XDM to represent the `World` which is un-instantiable from the XPDL.

Like Haskell's GHC (Glasgow Haskell Compiler), whether there really is a `World` that is present in the application at execution time or not is an implementation detail. Certainly it is crucial that the `World` is threaded through the chain of IO actions at evaluation time to ensure ordering, but implementations are free to optimise the world away as long as they preserve ordering.

4.1.2. Typing a Task

Ultimately we adopted the name *Task* instead of *IO* to represent our embracement of more than just I/O.

The first version of our Task Module was developed around the type definition of a `Task` as:

```
declare function task:task($realworld as element(adt:realworld))
                    as item()+
```

We quickly realised that using this module led to verbose syntax, and that the function syntax obscured the ordering of chains; the ordering of task execution being the most deeply nested and then extending outwards:

```
task:fmap(
    task:fmap(
        task:value("hello"),
        upper-case#1
    ),
    concat(?, " adam")
)
```

Figure 1. Example of Tasks using Function based syntax

To provide a more natural imperative syntax, we realised that instead of modelling a Task as a function type, we could model it as an XDM Map of functions which can be *applied*. An XDM Map is itself a function from its key to its value. By modelling a Task as Map, we could use the encapsulation concept from OOP (Object Oriented Programming) to place functions in the Task (Map), that act upon that task. Each function that we previously defined that operated upon a Task, we recreated as a function inside the Map which operates on the Task represented by the Map. Thus yielding a fluent imperative-like API that utilises the Map Lookup Operator to appear more familiar to imperative programmers:

```
task:value("hello")
    ? fmap(upper-case#1)
    ? fmap(concat(?, " adam"))
    ? RUN-UNSAFE()
```

Figure 2. Example of Tasks using fluent imperative-like syntax

So our Task type is finalised as:

```
map(xs:string, function(*))
```

More specifically our Task Map is defined as:

```
map {
    'apply': as function(element(adt:realworld)) as item()+,
    'bind': as function($binder as function(item()*) as map(xs:string,
function(*))) as map(xs:string, function(*)),
    'then': as function($next as map(xs:string, function(*))) as
map(xs:string, function(*)),
    'fmap': as function($mapper as function(item()*) as item()*) as
map(xs:string, function(*)),
    'sequence': as function($tasks as map(xs:string, function(*))+) as
map(xs:string, function(*)),
    'async': as function() as map(xs:string, function(*)),
    'catch': as function($catch as function(xs:QName?, xs:string,
map(*)) as map(xs:string, function(*))) as map(xs:string, function(*)),
    'catches': as function($codes as xs:QName*, $handler as
function(xs:QName?, xs:string, map(xs:QName, item()*)?) as item()*) as
map(xs:string, function(*)),
    'catches-recover': as function($codes as xs:QName*, $handler as
function() as item()*) as map(xs:string, function(*)),
    'RUN-UNSAFE': as function() as item()*
}
```

Observe that the `apply` entry inside the Task map retains our original Task type. The Map provides us with encapsulation which allows for the creation of an imperative-like API. By refactoring our existing Task functions we have been able to preserve both the function syntax-like API and the fluent imperative-like API. This provides developers the opportunity to choose whichever best suits their needs, or to work with a mix of syntaxes as appropriate to them.

4.1.3. Asynchronous Tasks

We provide a mechanism which explicitly allows the developer to state that a Task could benefit from being executed asynchronously. The `task:async` function allows the developer to state their intention, however EXPath Tasks does not specify whether, how, or if this actually executes asynchronously. This gives processors the ability to make informed decisions about concurrent execution based on input from the developer, but great freedom in how that is actually executed. The only constraint on implemetations is that the order of execution within a task chain must be preserved. Developers should rather think of `task:async` as providing a hint to the processor that asynchronous execution would be beneficial, rather than assuming asynchronous execution will always take place.

Conversely, as the only constraint that we place on implementers is that the order of execution within a task chain must be preserved, compliant processors are free to implicitly parallelise operations at execution time providing that constraint holds.

4.1.4. Executing a Task

Recall that a Haskell application starts with a `main` that must return an IO, thus framing the entire application as an IO action. The result of executing an XPDL is always an instance of the XDM (and possibly a PUL). Whilst we could certainly return a Task (map) as the result of the evaluation of our XPDL, what should the processor do when it encounters it? If the processor decides to serialize the XDM then we are likely at the mercy of the W3C XSLT and XQuery Serialization specification, which certainly won't execute our Task by *applying* it to transform the state of the world.

Three potential solutions that present themselves from our research are:

- Prescribe in the specification of EXPath Tasks that an implementation must execute a Task which is returned as the result of the XPDL in a certain manner.
- Incorporate the concept of a PUL into the specification of EXPath Tasks. Each Task would create an Update Primitive which is added into the PUL. The result of evaluating the XPDL would then be both an XDM and a PUL.
- Provide an explicitly unsafe function for evaluating a Task, similar to Haskell's `unsafePerformIO` or Monix Tasks's `runUnsafeSync`.

We decided to adopt a hybrid approach. We provide a `task:RUN-UNSAFE` function, where we explicitly prescribe that this should only appear *once* within an XPDL program, and that it *must* occur at the edge of the program, i.e. as the main function. However, we also explicitly state that implementers are free to override this function. For example, implementations that already support an XQuery Update PUL, may choose to promote a Task chain to a set of Update Primitives when this function is evaluated.

4.2. Using EXPath Tasks

We provide several examples to demonstrate key features of EXPath Tasks.

4.2.1. Composing Tasks

We can use monadic composition to safely compose together several tasks that may at execution time cause side effects, but at evaluation time result in an ordered chain of tasks.

Example 1. Safely Uppercasing a file

```
task:value("/tmp/my-file")
    ?fmap(file:read-text#1)
    ?fmap(fn:upper-case#1)
    ?fmap(fn:write-text("/tmp/my-file-upper", ?))
```

Consider the code in Example 1. We use the EXPath File Module to read the text of a file, we then upper-case the text, and finally write the text out to a new file. We start with a pure value Task holding the path of the source file, by *mapping* this through the `read-text` function a second new task is created. At evaluation time nothing has been executed, instead we have a task that describes that first there is a file path, and then secondly we should read a file from that path. We have composed two operations into one operation which preserves the ordering of the original operations. We then continue by *mapping* through the `upper-`

`case`, which composes another new task representing all three operations (file path, read-text, and upper-case) in order. Our last mapping composition results in a final new task which represents all four operations in order. When this final task is executed at runtime, each of the four operations will be performed in the correct order.

Through using the EXPath Tasks module, we have safely contained the side effects of the functions from the EXPath File Module, by deferring them from evaluation time to execution time. As the Task is a state transformation, we have also threaded the *World* through our task chain, which ensures that any XPDL processor must execute them in the correct order even in the face of aggressive optimisation.

4.2.2. Using Asynchronous Tasks

We can lift a Task to an Asynchronous Task, which can help provide the XPDL processor with hints about how best to parallelise an XPDL application.

The following is a refactored version of the *fork-join* example from xq-promise [35], to show how concurrent programming can be structured safely using EXPath Tasks.

The example performs 25 HTTP requests to 5 distinct servers and returns the results. First we show the synchronous version:

Example 2. Synchronous HTTP Fetching

```
let $tasks :=
    for $uri in ((1 to 5) !
      ('http://www.google.com', 'http://www.yahoo.com',
       'http://www.amazon.com', 'http://cnn.com',
       'http://www.msnbc.com'))
    let $task :=
        task:value($uri)
            ?fmap(http:send-request(<http:request method="GET" />, ?))
            ?fmap(fn:tail#1)
            ?fmap(fn:trace(?, 'Results found: '))
            ?fmap(function ($res) {
                $res//*:a[@href => matches('^http')]
            })
return
    task:sequence($tasks)
        ?RUN-UNSAFE()
```

Now we show the asynchronous version, where we have only needed to insert two lines of code, the call to `task:async` which lifts each Task into an Asynchronous Task, and a binding to `task:wait-all`:

Example 3. Asynchronous HTTP Fetching

```
let $tasks :=
    for $uri in ((1 to 5) !
      ('http://www.google.com', 'http://www.yahoo.com',
       'http://www.amazon.com', 'http://cnn.com',
       'http://www.msnbc.com'))
    let $task :=
        task:value($uri)
            ?fmap(http:send-request(<http:request method="GET" />, ?))
            ?fmap(fn:tail#1)
            ?fmap(fn:trace(?, 'Results found: '))
            ?fmap(function ($res) {
                $res//*:a[@href => matches('^http')]
            })
            ?async()
return
    task:sequence($tasks)
        ?bind(task:wait-all#1)
        ?RUN-UNSAFE()
```

4.2.3. Using Tasks with IXSL

We now consider how Tasks could be used within an IXSL stylesheet for a Saxon-JS web application. Here we use Tasks to enable both concurrency (an asynchronous HTTP request) and side effects (HTML DOM updates). The code in Example 4 shows an IXSL event handling template for onclick events for the "go" button, and associated functions. The main action of clicking the "go" button is to send an asynchronous HTTP request. The intention is that the HTTP response will provide new content for the `<div id="target">` element in the HTML page, as directed by the local `f:handle-http-response` function. But while awaiting the HTTP response, the "target" `div` is first updated to provide a "Request processing..." message, and the "go" button is hidden; as directed by the local `f:onclick-page-updates` function.

Example 4. Asynchronous HTTP using Tasks in IXSL

```
   <xsl:template match="button[@id eq 'go']" mode="ixsl:onclick">
      <xsl:variable name="onclick-page-updates-task"
                    select="task:of(f:onclick-page-updates#0)"/>
      <xsl:variable name="http-post-task"
                    select="task:of(function(){http:post($request-body,
$request-options)})"/>
      <xsl:variable name="async-http-task"
```

```
                  select="$http-post-task ? fmap(f:handle-http-
response#1) ? async()"/>
      <xsl:sequence select="task:RUN-UNSAFE(task:then($onclick-page-
updates-task, $async-http-task))"/>
   </xsl:template>

   <xsl:function name="f:onclick-page-updates">
      <ixsl:set-style name="display" select="'none'"
                     object="ixsl:page()//button[id='go']"/>
      <xsl:result-document href="#target" method="ixsl:replace-content">
         <p>Request processing...</p>
      </xsl:result-document>
   </xsl:function>

   <xsl:function name="f:handle-http-response">
      <xsl:param name="response" as="map(*)"/>
      <xsl:for-each select="$response?body">
         <xsl:result-document href="#target"
                              method="ixsl:replace-content">
            <p>Response from request:</p>
            <xsl:sequence select="."/>
         </xsl:result-document>
      </xsl:for-each>
      <ixsl:set-style name="display" select="'inline'"
                     object="ixsl:page()//button[id='go']"/>
   </xsl:function>
```

Through using the EXPath Tasks module, we have safely contained the side effects of the local functions. Meanwhile, the use of the `task:async` function allows the Saxon-JS processor to use an asynchronous implementation of the EXPath HTTP Client 2.0 `http:post` function. The task chain is created making use of `task:fmap` to pass the HTTP response to the handler function; and `task:then` to compose the initial `$onclick-page-updates-task` with the main `$async-http-task`, ensuring the correct order for their side effects.

5. Conclusion

In this paper we have surveyed the current state-of-the-art mechanisms by which XPDL processors allow side effects and concurrent programming, and the options available to non-XPDLs for managing side effects and providing concurrent or parallel programming. From this research we have then developed and specified EXPath Tasks, a module of XPath extension functions, that allow developers to safely encapsulate side-effecting functions so that at evaluation time they appear as pure functions and enforce the expected order of execution. Finally, we have developed several reference implementations of EXPath Tasks to demonstrate the feasability of implementing our specification.

Were the necessary functions available for performing node updates, we believe that the IO Monad approach taken by EXPath Tasks could even have benefits over using XQuery Update. Whilst it provides similarly strong deferred semantics like a PUL, a *PUL* is completely opaque, and one cannot compute over it, unlike a Task chain where Tasks may be composed together.

Whilst at a casual glance it may appear that EXPath Tasks have some similarities to xq-promise, we should be careful to point out that they work quite differently in practice. We believe that EXPath Tasks has the following advantages over xq-promise:

- Correct Ordering of Execution.

  Under aggressive functional optimisation, EXPath Tasks will still preserve the correct order of execution even when tasks have no explicit dependency between them. EXPath Tasks can guarantee the order because they transparently thread the *World* through the chain of computation as tasks are composed, which implicitly introduces dependencies between the Tasks.

- Flexible Asynchronous Processing.

  The asynchronous processing model of EXPath Tasks is very generalised, and only makes guarantees about ordering of execution. This enables many forms of concurrent programming to be expressed using EXPath Tasks, whereas xq-promise only offers *fork-join*. In fact xq-promise can easily be reimplemented atop EXPath tasks, including *fork-join*:

  ```
  declare function local:fork-join($tasks as task:Task(~An)+)
          as task:Task(array(~An)) {
      task:sequence($tasks ! task:async#1)
          ?bind(task:wait-all#1)
  };
  ```

  Interestingly, if the xq-promise API were reimplemented atop EXPath Tasks, it would gain stronger guarantees about execution order.

  Likewise our generalised approach, whilst making explicit the intention of parallelism, does not restrict processors from making further implicit parallelisation optimisations.

- Potential Performance

  An xq-promise Promise is a deferred computation that cannot be executed until its `fork-join` function is called. In comparison EXPath Tasks's Asynchronous Tasks can begin execution at runtime as soon as their construct function is executed, thus making better use of computer resources by starting computation earlier than would be possible in xq-promise.

It will certainly be interesting to see how the XML community responds to our EXPath Tasks specification. We are hopeful that developers working with Tasks need not necessarily have any understanding of Monads to be able to fully exploit the benefits of EXPath Tasks.

We are still at an early stage of investigating how well use of the Task module can be incorporated into IXSL stylesheets for Saxon-JS applications. Does the Task module provide a good solution for handling asynchronous processing and side effects in Saxon-JS? This may only be answerable once more examples have been trialled, once the Saxon-JS implementation is more advanced.

Given an existing Saxon-JS application, a move to use the Task module could involve a significant amount of restructuring. To use the Task module properly, all side-effecting expressions should be wrapped in tasks, and care would need to be taken to chain them together appropriately. Side-effecting expressions are likely to be found in numerous different templates, and so bringing the tasks together could be a challenge, and would likely involve considerable redesign. These challenges are not necessarily a problem with the Task module, but given that currently developers can be relatively free with how side effects and asynchronous processes fit into their XSLT programs; the move to any solution which requires explicit strict management of these is going to be a fairly radical change. But this work would not be without benefit: the current lack of management of side effects can easily result in unexpected results if the developer is not careful. The use of Tasks would eliminate this risk.

Further work is also required to work out exactly how to use Tasks to accomplish some specific actions within a Saxon-JS application. For example, providing a mechanism which allows a user to abort an asynchronous HTTP request. Combining the use of Tasks with IXSL event handling templates, does not seem to work. Instead it seems a solution requires another way to create event listeners from within the XSLT; in which case, perhaps new IXSL extensions are needed.

5.1. Future Work

We have identified several areas for possible future research:

- Stronger/Stricter Explicit Typing

  The explicit types we have specified in our Task Module are not as strict as we would like. This is in general due to a lack of a stronger type system which would allow us to express both abstract and generic types. At run-time the correct types will be inferred by the processor. It would be interesting to research modifications to the XDM so that we can statically express stricter types. For instance, the Saxon processor provides the tuple type [64] syntax extension as a way of defining a more precise type for maps.

  We recognise there may also be an approach where function generation is used, to generate Task functions with stricter types by type switching on incoming parameters. Due to the large number of types in the XDM to switch over, such generation would itself likely need to be computed.

- Side effects between Concurrent Tasks

  We have provided no mechanisms for avoiding side effects across shared state between parallel tasks at execution time, e.g. race conditions, data corruption, etc. Often such issues can be avoided by developers decomposing asynchronous tasks into smaller asynchronous tasks which have to synchronize via `task:wait-all`, and then begin asynchronously again. A set of functional Task based synchronization primitives which could be used to help in parallel situations would be an interesting extension.
- Additional convenience functions

  Whilst we have provided the building blocks necessary for general computation, additional convenience functions could be added. For instance `gather` (similar to `task:sequence` but with relaxed ordering), `withAsync` (which lifts a normal function into an Asynchronous Task), and `parZip` (which asynchronously zips the results of two tasks together).

  We have provided mechanisms for working with XPath errors, however we could also consider functions for working with *error values*. We see no reason why something akin to an Either (disjoint union) could not be developed to work with EXPath Tasks, where a result is *either* an error value or the result of successful computation.

A. EXPath Tasks Module Definitions

A.1. Namespaces and Prefixes

This module makes use of the following namespaces to contain its application. The URIs of the namespaces and the conventional prefixes associated with them are:

- `http://expath.org/ns/task` for functions -- associated with `task`.
- `http://expath.org/ns/task/adt` for abstract data types -- associated with `adt`.

A.2. Types

As an attempt at simplifying the written definition of the functions within the Task Module, we have specified a number of type aliases. The concrete types are likely of little interest to users of the Task Module who are more concerned with behaviour than implementation detail. Implementers which need such detail may substitute the aliases for the concrete types as defined below.

We have followed the XPath convention of using lower-cased names for our functions, apart from `task:RUN-UNSAFE` where the use of continuous capital letters is intended to draw developer attention. Our type aliases are described using a capitalised-cased naming convention to visually distinguish them from function names.

Alias	**Concrete Type**
~A	The `~` signifies that this is a generic type, and the `A` is just a placeholder for the actual type. Concretely this is at least an `item()*`, however intelligent processors can likely infer and enforce stricter types through the functionally composed Task chain.

Alias	**Concrete Type**

task:Task(~A)	The `task:Task` type alias, is concretely `map(xs:string, function(*))`.
	The inner aliased generic type, indicates that the Task when executed returns a result of type `~A`.
	Specifically the Task map has the following non-optional entries:
	
	`map {`
	    `'apply': as function(World) as item()+,`
	    `'bind': as function($binder $binder as function(~A) as task:Task(~B)) as task:Task(~B),`
	    `'then': as function($next as task:Task(~B)) as task:Task(~B),`
	    `'fmap': as function($mapper as function(~A) as ~B) as task:Task(~B),`
	    `'sequence': as function($tasks as task:Task(~An)+) as task:Task(array(~An)),`
	    `'async': as function() as task:Task(task:Async(~A)),`
	    `'catch': as function($catch as function(xs:QName?, xs:string, map(*)) as task:Task(~B)) as task:Task(~B),`
	    `'catches': as function($codes as xs:QName*, $handler as function(xs:QName?, xs:string, map(xs:QName, item()*)?) as ~B) as task:Task(~B),`
	    `'catches-recover': as function($codes as xs:QName*, $handler as function() as ~B) as task:Task(~B),`
	    `'RUN-UNSAFE': as function() as ~A`
	`}`
	
	Note: Each of the functions defined in the Task Map have the exact same behaviour as their cousins of the same name residing outside of the map. The only difference is that the functions inside the Map don't need an explicit task argument.

Alias	**Concrete Type**

task:ErrorObject	The `task:ErrorObject` type alias, is concretely `map(xs:QName, item()*)`.
	All entries in the map are optional, but otherwise it is structured as:
	
	`map {`
	`    xs:QName("err:value") : item()*,`
	`    xs:QName("err:module") : xs:string?,`
	`    xs:QName("err:line-number") : xs:integer?,`
	`    xs:QName("err:column-number") : xs:integer?`
	`    xs:QName("err:additional") : item()*`
	`}`

task:Async(~A)	The `task:Async` type alias, is concretely `function(element(adt:scheduler)) as ~A`. The inner aliased generic type, indicates that the Async if it runs to completion will compute a result of type `~A`.

A.3. Functions

A.3.1. Basic Task Construction

This group of functions offer facilities for constructing basic tasks. They usually form the starting point of a task chain.

A.3.1.1. task:value

Summary	Constructs a Task from a *pure* value.
Signature	`task:value($v as ~A) as task:Task(~A).`
Rules	When the task is run it will return the value of *$v*.
Notes	In Haskell this would be known as `return` or sometimes alternatively `unit`. In Scala Monix this would be known as `now` or `pure`. In formal descriptive terms this is: `value :: a -> Task a`

Example **Example A.1. Task from a String**

```
task:value("hello world")
```

A.3.1.2. task:of

Summary
: Constructs a Task from a function.

  This provides a way to wrap a potentially non-pure (i.e. side-effecting) function and delay its execution until the Task is executed.

Signature
: `task:of($f as function() as ~A) as task:Task(~A).`

Rules
: The function is lifted into the task, which is to say that the function will not be executed until the task is executed. When the task is run, it will execute the function and return its result.

Notes
: In Haskell there is no direct equivalent.

  In Scala Monix this would be known as `eval` or `delay`.

  In formal descriptive terms this is:

```
of :: (() -> a) -> Task a
```

Example **Example A.2. Task which computes the system time from a side-effecting function.**

```
task:of(util:system-time#0)
```

A.3.2. Task Composition

This group of functions offer facilities for functionally composing tasks together.

A.3.2.1. task:bind

Summary
: Composes a new Task from an existing task and a binder function which creates a new task from the existing task's value.

Signature
: `task:bind($task as task:Task(~A), $binder as function(~A) as task:Task(~B)) as task:Task(~B).`

Rules
: When the resultant task is executed, the binder function processes the existing task's value, and then the result of the task is returned.

Notes
: In Haskell this is also called `bind` and often written as `>>=`.

  In Scala Monix this is known as `flatMap`.

In formal descriptive terms this is:

```
bind :: Task a -> (a -> Task b) -> Task b
```

Examples

Example A.3. Using bind to Square a number

```
task:bind(task:value(99), function($v) {
    task:value($v * $v)
})
```

Example A.4. Using bind to Transform a value

```
task:bind(task:value("hello"), function($v) {
    task:value(fn:upper-case($v))
})
```

Example A.5. Using bind to conditionally raise an error

```
task:bind(task:value("hello"), function($v) {
    if ($v eq "goodbye")
    then
        task:error((), "It's not yet time to say goodbye!",
())
    else
        task:value($v)
})
```

Example A.6. Using bind to compose two tasks

```
let $task1 := task:value("hello")
let $task2 := task:value("world")
return
    task:bind($task1, function($v1) {
        task:bind($task2, function($v2) {
            task:value($v1 || " " || $v2)
        })
    })
```

A.3.2.2. task:then

Summary
: Composes a new Task from an existing task and a new task. It is similar to `task:bind` but discards the existing task's value.

Signature
: `task:then($task as task:Task(~A), $next as task:Task(~B)) as task:Task(~B).`

Rules
: When the resultant task is executed, the existing task is executed and the result discarded, and then the result of the next task is returned.

  `task:then($task, $next)` is equivalent to `task:bind($task, function($_) { $next })`.

Notes
: In Haskell this is also a form of bind which is sometimes called `then`, and often written as `>>`.

  In Scala Monix this is direct equivalent.

  In formal descriptive terms this is:

```
then :: Task a -> (_ -> Task b) -> Task b
```

Example
: **Example A.7. Sequentially composing two tasks**

```
task:then(task:value("something we don't further need"),
task:value("something important"))
```

A.3.2.3. task:fmap

Summary
: Composes a new Task from an existing task and a mapping function which creates a new value from the existing task's value.

Signature
: `task:fmap($task as task:Task(~A), $mapper as function(~A) as ~B) as task:Task(~B).`

Rules
: When the resultant task is executed, the mapper function processes the existing task's value, and then the result of the task is returned.

Notes
: In Haskell this is also called `fmap` and often written as `<$>`.

  In Scala Monix this is known as `map`.

  In formal descriptive terms this is:

```
fmap :: Task a -> (a -> b) -> Task b
```

Examples

Example A.8. Upper-casing a Task String

```
task:fmap(task:value("hello"), fn:upper-case#1)
```

Example A.9. Concatenating a Task String

```
task:fmap(task:value("hello"), fn:concat(?, " world"))
```

Example A.10. Extracting the code-points of a Task String (e.g. type conversion, String to Integer+)

```
task:fmap(task:value("hello"), fn:string-to-codepoints#1)
```

A.3.2.4. task:sequence

Summary	Constructs a new Task representating the sequential application of one or more other tasks.
Signature	`task:sequence($tasks as task:Task(~An)+) as task:Task(array(~An))`.
Rules	When the resultant task is executed, each of the provided tasks will be executed sequentially, and the results returned as an XDM array. The order of entries in the resultant array is the same as the order of *$tasks*.
Notes	In Haskell and Scala Monix this is known as `sequence`. In formal descriptive terms this is: `sequence :: [Task a] -> Task [a]`

Examples

Example A.11. Sequencing three Tasks into one

```
task:sequence((task:value("hello"), task:value(54),
task:value("goodbye"))
```

A.3.3. Task Error Management

This group of functions offers facilities for using tasks in the face of XPath errors. Several can be used along with `task:error` as a form of conditional branching or downward flow control.

A.3.3.1. task:error

Summary
: Constructs a Task that raises an error.

This is a Task abstraction for `fn:error`.

Signature
: `task:error($code as xs:QName?, $description as xs:string, $error-object as task:ErrorObject?) as task:Task(none).`

Rules
: The error is not raised until the task is run.

The parameters *$code,* and *$description* have the same purpose as those with the same name defined for `fn:error`.

The parameter *$error-object* has the same purpose but is a type restriction of the parameter with the same name defined for `fn:error`, it should be of type task:ErrorObject.

Notes
: In Haskell this would be closest to `fail`.

In Scala Monix this would be known as `raiseError`.

In formal descriptive terms this is:

```
error :: (code, description, error-object) -> Task none
```

Examples

Example A.12. Constructing a simple Task Error

```
task:error(xs:QName("local:error001"), "BOOM!", ())
```

A.3.3.2. task:catch

Summary
: Constructs a Task which catches any error raised by another task.

This is similar to `task:catches` except that all errors are caught.

Signature
: `task:catch($task as task:Task(~A), $handler as function(xs:QName?, xs:string, task:ErrorObject?) as task:Task(~B)) as task:Task(~B).`

Rules
: When the resultant task is executed, the handler function catches any error from executing the existing task, and then the result of the handler task is returned.

The handler function accepts three arguments, the first is the QName of the error that was caught, the second is the description of the error that was caught, and the third are the ancillary error details collected as a `task:ErrorObject`.

If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes

In Haskell this is similar to `catch`.

In Scala Monix this would be similar to `onErrorHandleWith`.

In formal descriptive terms this is:

```
catches :: Task a -> ([code, description, errorObject] -> Task b) -
> Task b
```

Example

Example A.13. Using catch to recover from an error

```
let $my-error-code := xs:QName("local:error01")
return
    task:catch(task:error($my-error-code, "Boom!", ()),
function($actual-code, $actual-description, $actual-error-
object) {
        "Handled error: " || $actual-code
    })
```

A.3.3.3. task:catches

Summary

Constructs a Task which catches specific errors of another task.

This is similar to `task:catch-recover` except that the error handler receives details of the error.

Signature

`task:catches($task as task:Task(~A), $codes as xs:QName*, $handler as function(xs:QName?, xs:string, task:ErrorObject?) as ~B) as task:Task(~B).`

Rules

When the resultant task is executed, the handler function catches any matching errors identified by the parameter *$codes* from executing the existing task, and then the result of the handler task is returned.

The handler function accepts three arguments, the first is the QName of the error that was caught, the second is the description of the error that was caught, and the third are the ancillary error details collected as a `task:ErrorObject`.

If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes

In Haskell this is similar to `catches`.

In Scala Monix this would be similar to `onErrorHandle`.

In formal descriptive terms this is:

```
catches :: Task a -> ([code] -> ([code, description, errorObject] ->
b)) -> Task b
```

Example **Example A.14. Using catches to recover from an error**

```
let $my-error-code := xs:QName("local:error01")
return
    task:catches(task:error($my-error-code, "Boom!", ()),
($my-error-code, xs:QName("err:XPDY004")), function($actual-
code, $actual-description, $actual-error-object) {
        "Handled error: " || $actual-code
    })
```

A.3.3.4. task:catches-recover

Summary Constructs a Task which catches specific errors of another task.

This is similar to `task:catches` except that the error handler does not receive details of the error.

Signature `task:catches-recover($task as task:Task(~A), $codes as xs:QName*, $handler as function() as ~B) as task:Task(~B).`

Rules When the resultant task is executed, the handler function catches any matching errors identified by the parameter *`$codes`* from executing the existing task, and then the result of the handler task is returned.

If no errors are raised by the existing task, the handler will not be called, and instead this task acts as an identity function.

Notes In Haskell this is similar to `catches`, but it does not pass the error details to the *`$handler`*.

In Scala Monix this would be similar to `onErrorRecover`.

In formal descriptive terms this is:

```
catches-recover :: Task a -> ([code] -> (\_ -> b)) -> Task b
```

Example **Example A.15. Using catches-recover to recover from an error**

```
let $my-error-code := xs:QName("local:error01")
return
    task:catches-recover(task:error($my-error-code,
"Boom!", ()), ($my-error-code), function() {
        "Recovering from error..."
        })
```

A.3.4. Asynchronous Tasks

This group of functions offers facilities for constructing asynchronous tasks and acting upon their progress.

A.3.4.1. task:async

Summary	Constructs an Asynchronous Task from an existing Task.
Signature	`task:async($task as task:Task(~A)) as task:Task(task:Async(~A))`.
Rules	The existing task will be composed into a new task which may be executed asynchronously. This function makes no guarantees about how, when, or if the asynchronous task is executed other than the fact that execution will not begin before the task itself is executed. Implementations are free to implement asynchronous tasks using any mechanism they wish including cooperative multitasking, preemptive multitasking, or even plain old single-threaded synchronous. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect. When the task is run, it may start an asynchronous process which executes the task, regardless it returns a reference to the (possibly) asynchronous process, which may later be used for cancellation or obtaining the result of the task. If the function call results in asynchronous behaviour (i.e. a fork of the execution path happens), then the asynchronous task inherits the *Static Context*, and a copy of the *Dynamic Context* where the *Context item*, *Context position*, and *Context size* have been reinitialised. If an implementation supports XQuery Update PUL, then any Update Primitives generated in the Asynchronous Task are merged back to the main Task only when `task:wait` or `task:wait-all` is employed.
Notes	In Haskell this is similar to `async` from the `Control.Concurrent.Async` package. In Scala Monix this would be known as `executeAsync`. In formal descriptive terms this is:

```
async :: Task a -> Task (Async a)
```

Example **Example A.16. Task which asynchronously posts a document**

```
task:async(
    task:fmap(
        task:value("http://somewebsite.com"),
        http:post(?, <some-document/>)
    )
)
```

A.3.4.2. task:wait

Summary Given an Async this function will extract its value and return a Task of the value.

Signature `task:wait($async as task:Async(~A)) as task:Task(~A).`

Rules At execution time of the task returned by this function, if the Asynchronous computation represented by the *`$async`* reference has not yet completed, then this function will block until the asynchronous computation completes.

This function makes no guarantees about how, when, or if blocking occurs other than the fact that any blocking (if required) will not begin before the task itself is executed.

Implementations are free to implement waiting upon asynchronous tasks using any mechanism they wish. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes In Haskell this is similar to `wait` from the `Control.Concurrent.Async` package.

In Scala Monix this would be similar to `Await.result`.

In formal descriptive terms this is:

```
wait :: Async a -> Task a
```

Example **Example A.17. Task waiting on an asynchronous task**

```
let $async-task :=
          task:async(
              task:fmap(
                  task:value("http://somewebsite.com"),
                  http:post(?, <some-document/>)
              )
          )
      return

          (: some further task chain of processing... :)

          (: wait on the asynchronous task to complete :)
          task:bind(
              $async-task,
              task:wait#1
          )
```

A.3.4.3. task:wait-all

Summary	Given multiple Asyncs this function will extract their values and return a Task of the values.
Signature	`task:wait-all($asyncs as array(task:Async(~A))) as task:Task(array(~A))`.
Rules	At execution time of the task returned by this function, if any of the Asynchronous computations represented by the *`$asyncs`* references have not yet completed, then this function will block until all the asynchronous computations complete.

This function makes no guarantees about how, when, or if blocking occurs other than the fact that any blocking (if required) will not begin before the task itself is executed.

Implementations are free to implement waiting upon asynchronous tasks using any mechanism they wish. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

This is equivalent to:

```
task:bind($task, function($asyncs as array(*)) as
map(xs:string, function(*)) {
    task:sequence(array:flatten(array:for-each($asyncs,
task:wait#1)))
})
```

Notes — In Haskell there is no direct equivalent, but it can be modelled by a combination of `wait` and `sequence`.

In Scala Monix there is no direct equivalent.

In formal descriptive terms this is:

```
wait-all :: [Async a] -> Task [a]
```

Example — **Example A.18. Task waiting on multiple asynchronous tasks**

```
let $async-tasks :=
    (
        task:async(
            task:fmap(
                task:value("http://websiteone.com"),
                http:post(?, <some-document/>)
            )
        ),
        task:async(
            task:fmap(
                task:value("http://websitetwo.com"),
                http:post(?, <some-document/>)
            )
        )
    )
return

    (: some further task chain of processing... :)

    (: wait for all asynchronous tasks to complete :)
    task:bind(
        task:sequence($async-tasks),
        task:wait-all#1
    )
```

A.3.4.4. task:cancel

Summary	Given an Async this function will attempt to cancel the asynchronous process.
Signature	`task:cancel($async as task:Async(~A)) as task:Task().`
Properties	This function is *non-blocking*.
Rules	At execution time of the task returned by this function, cancellation of the Asynchronous computation represented by the *`$async`* reference may be attempted.

This function makes no guarantees about how, when, or if cancellation occurs other than the fact that any cancellation (if required/possible) will not begin before the task itself is executed. Regardless the Asynchronous reference is invalidated by this function.

Implementations are free to implement cancellation of asynchronous tasks using any mechanism they wish, they are also free to ignore cancellation as long as the Asynchronous reference is still invalidated. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes

In Haskell this is similar to `cancel` from the `Control.Concurrent.Async` package.

In Scala Monix this is known as `cancel`.

In formal descriptive terms this is:

```
cancel :: Async a -> Task ()
```

Example

Example A.19. Cancelling an asynchronous task

```
let $async-task :=
    task:async(
        task:fmap(
            task:value("http://somewebsite.com"),
            http:post(?, <some-document/>)
        )
    )
return

    (: some further task chain of processing... :)

    (: cancel the asynchronous task :)
    task:bind(
        $async-task,
        task:cancel#1
    )
```

A.3.4.5. task:cancel-all

Summary

Given multiple Asyncs this function will attempt to cancel all of the asynchronous processes.

Signature `task:cancel-all($asyncs as array(task:Async(~A))) as task:Task()`.

Properties This function is *non-blocking*.

Rules At execution time of the task returned by this function, cancellation of all Asynchronous computations represented by the *$asyncs* references may be attempted.

This function makes no guarantees about how, when, or if cancellation occurs other than the fact that any cancellation (if required/possible) will not begin before the task itself is executed. Regardless the Asynchronous references are invalidated by this function.

Implementations are free to implement cancellation of asynchronous tasks using any mechanism they wish, they are also free to ignore cancellation as long as the Asynchronous references are still invalidated. The only restriction on implementations is that the processing order of task chains and asynchronous task chains must be preserved, so that the user gets the result that they should expect.

Notes In Haskell there is no direct equivalent, but it can be modelled by a combination of `cancel` and `sequence`.

In Scala Monix there is no direct equivalent.

In formal descriptive terms this is:

```
cancel-all :: [Async a] -> Task ()
```

Example **Example A.20. Cancelling asynchronous tasks**

```
let $async-tasks :=
    (
        task:async(
            task:fmap(
                task:value("http://websiteone.com"),
                http:post(?, <some-document/>)
            )
        ),
        task:async(
            task:fmap(
                task:value("http://websitetwo.com"),
                http:post(?, <some-document/>)
            )
        )
    )
```

```
return

    (: some further task chain of processing... :)

    (: cancel all asynchronous tasks :)
    task:bind(
        task:sequence($async-tasks),
        task:cancel-all#1
    )
```

A.3.5. Unsafe Tasks

This defines a single function `task:RUN-UNSAFE`, which is useful only when a task chain needs to be executed. If an XPDL implementation cannot provide a better mechanism, then this may be implemented and used as a last resort.

A.3.5.1. task:RUN-UNSAFE

Summary	Executes a Task Chain and returns the result. This function is **inherently unsafe**, as it causes any side effects within the Task chain to be actualised. If this function is used within an application, it should only be invoked once, and it should be at the edge of the application, i.e. in the position where it is the first and only thing to be directly executed by the application at runtime. No further computation, neither on the result of this function, or after this function call should be attempted by the application.
Signature	`task:RUN-UNSAFE($task as task:Task(~A)) as ~A`.
Properties	This function is *nondeterministic*.
Rules	At execution time, the task chain is evaluated and the result returned. However, if implementations can provide a safer mechanism for the execution of a Task after the XPDL has completed evaluation, then they are free to override this as they see fit. Once such mechanism could be to promote the Task chain to a set of Update Primitives within a PUL and then demote this to an identity function.
Notes	In Haskell the closest equivalent is `unsafePerformIO`. In Scala Monix the closest approach would be a combination of `runToFuture` and `Await.result`. In formal descriptive terms this is:

```
RUN-UNSAFE :: Task a -> a
```

Example

Example A.21. Unsafely executing a Task

```
(:~
 : Just a utility function for calculating
 : previous sightings of Halley's comet
 :)
declare function local:halleys-sightings($before-year) {
    let $start := 1530
    let $interval := 76

    for $range in
        ($start - $interval to $before-year - $interval)
    let $visible := $range + $interval
    where (($visible - $start) mod $interval) eq 0
    return
        $visible
};

let $task := task:fmap(
    task:fmap(
        task:of(util:system-time#0),
        fn:year-from-date#1
    ),
    local:halleys-sightings#1
)
return

    task:RUN-UNSAFE($task)
```

Bibliography

[1] James Clark. Steve DeRose. *XML Path Language (XPath) Version 1.0*. W3C Recommendation 16 November 1999 (Status updated October 2016). 1999-11-16. https://www.w3.org/TR/1999/REC-xpath-19991116/.

[2] Anders Berglund. Scott Boag. Mary Fernández. Scott Boag. Michael Kay. Jonathan Robie. Jérôme Siméon. *XML Path Language (XPath) 2.0 (Second Edition)*. W3C Recommendation 14 December 2010 (Link errors corrected 3 January 2011; Status updated October 2016). 2010-12-14. https://www.w3.org/TR/xpath20/.

[3] Mary Fernandez. K Karun. Mark Scardina. *XPath Requirements Version 2.0*. W3C Working Draft 3 June 2005. 2005-06-03. https://www.w3.org/TR/xpath20req/.

[4] Denise Draper. Peter Fankhauser. Mary Fernández. Ashok Malhotra. Kristoffer Rose. Michael Rys. Jérôme Siméon. Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*. W3C Recommendation 14 December 2010 (revised 7 September 2015). 2015-09-07. https://www.w3.org/TR/xquery-semantics/.

[5] Steve Muench. Mark Scardina. *XSLT Requirements Version 2.0*. W3C Working Draft 14 February 2001. 2001-02-14. https://www.w3.org/TR/xslt20req/.

[6] Don Chamberlin. Peter Fankhauser. Massimo Marchiori. Jonathan Robie. *XML Query (XQuery) Requirements*. W3C Working Group Note 23 March 2007. 2007-03-27. https://www.w3.org/TR/xquery-requirements/.

[7] John Snelson. Jim Melton. *XQuery Update Facility 3.0. Pending Update Lists*. W3C Working Group Note 24 January 2017. 2017-01-24. https://www.w3.org/TR/xquery-update-30/#id-pending-update-lists.

[8] Christian Grün. BaseX. 2018-10-31T16:11:00Z. *Jobs Module*. BaseX. http://docs.basex.org/wiki/Jobs_Module.

[9] Adam Retter. eXist-db. *eXist-db Util XQuery Module*. Git Hub. http://www.exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/util&location=java:org.exist.xquery.functions.util.UtilModule&details=true.

[10] Adam Retter. eXist-db. *eXist-db Scheduler XQuery Module*. Git Hub. http://www.exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/scheduler&location=java:org.exist.xquery.modules.scheduler.SchedulerModule.

[11] IBM. 2012-03-07. *IBM100 - Power 4 : The First Multi-Core, 1GHz Processor*. https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

[12] Michael Perrone. 2009. *Multicore Programming Challenges*. IBM, TJ Watson Research Lab. 978-3-642-03868-6. 10.1007/978-3-642-03869-3_1. Springer. https://link.springer.com/chapter/10.1007%2F978-3-642-03869-3_1. *Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science*. 5704.

[13] Pedro Fonseca. Cheng Li. Rodrigo Rodrigues. 2011-04-10. *Finding complex concurrency bugs in large multi-threaded applications*. *EuroSys '11 Proceedings of the sixth conference on Computer systems*. 215-228. ACM. 978-1-4503-0634-8. 10.1145/1966445.1966465. https://dl.acm.org/citation.cfm?id=1966465.

[14] Matthew Loring. Mark Marron. Daan Leijen. 2017-10-24. *Semantics of Asynchronous JavaScript*. *DLS 2017 Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages table of contents*. 51-62. ACM. 978-1-4503-5526-1. 10.1145/3133841.3133846. https://dl.acm.org/citation.cfm?id=3133846.

[15] Cosmin Radoi. Stephan Herhut. Jaswanth Sreeram. Danny Dig. 2015-01-24. *Are Web Applications Ready for Parallelism?*. *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 289-290. ACM. 978-1-4503-3205-7. 10.1145/2688500.2700995. https://dl.acm.org/citation.cfm?id=2700995.

[16] Henry G. Baker, Jr.. Carl Hewitt. 1977-08-15. *The Incremental Garbage Collection of Processes*. *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. 55-59. ACM. 10.1145/800228.806932.

[17] Peter Hibbard. 1976. *Parallel Processing Facilities*. *New Directions in Algorithmic Languages*. 1-7.

[18] Daniel Friedman. David Wise. 1976. *The Impact of Applicative Programming on Multiprocessing*. *International Conference on Parallel Processing 1976*. 263–272. ACM.

[19] Jeffrey Dean. Sanjay Ghemawat. 2004. *MapReduce: Simplified Data Processing on Large Clusters*. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. 137-150. https://research.google.com/archive/mapreduce-osdi04.pdf.

[20] Michael Kay. 2015-02-14. *Parallel Processing in the Saxon XSLT Processor*. *XML Prague 2015 Conference Proceedings*. 978-80-260-7667-4. http://www.saxonica.com/papers/xmlprague-2015mhk.pdf.

[21] Jonathan Robie. 2016-03-03T12:05:03-05:00. EXPath Mailing List. *Re: [expath] Re: New Modules? Promise Module, Async Module*. https://groups.google.com/forum/#!msg/expath/Isjeez-5op4/-DCn-KJGBAAJ.

[22] O'Neil Delpratt. Michael Kay. 2013-08-06. *Interactive XSLT in the browser*. *Balisage Series on Markup Technologies, vol. 10 (2013)*. 10. https://doi.org/10.4242/BalisageVol10.Delpratt01. https://www.balisage.net/Proceedings/vol10/html/Delpratt01/BalisageVol10-Delpratt01.html.

[23] Adam Retter. 2018-10-03. *EXPath and Asynchronous HTTP*. https://blog.adamretter.org.uk/expath-and-asynchronous-http/.

[24] Jesús Camacho-Rodríguez. Dario Colazzo. Ioana Manolescu. 2015. *PAXQuery: Efficient Parallel Processing of Complex XQuery*. *IEEE Transactions on Knowledge and Data Engineering*. Institute of Electrical and Electronics

Engineers. 1977-1991. 10.1109/TKDE.2015.2391110. https://hal.archives-ouvertes.fr/hal-01162929/document.

[25] Ghislain Fourny. Donald Kossmann. Markus Pilman. Tim Kraska. Daniela Florescu. Darin Mcbeath. *WWW 2009 MADRID! Track: XML and Web Data / Session: XML Querying XQuery in the Browser*. 2009-04-20. *XQuery in the Browser*. http://www2009.eprints.org/102/1/p1011.pdf.

[26] Philip Fennell. 2013-06-15. *XML London 2013 Conference Proceedings*. 1. 978-0-9926471-0-0. *Extremes of XML*. https://xmllondon.com/2013/xmllondon-2013-proceedings.pdf#page=80.

[27] Jirka Kosek. John Lumley. 2013-12-03. *Binary Module 1.0*. EXPath. http://expath.org/spec/binary/1.0.

[28] Christian Grün. 2015-02-20. *File Module 1.0*. EXPath. http://expath.org/spec/file/1.0.

[29] BaseX. 2018-08-26T16:13:04Z. *Concepts: Pending Update List*. BaseX. http://docs.basex.org/wiki/XQuery_Update#Pending_Update_List.

[30] MarkLogic. 2018. *xdmp:spawn — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/xdmp:spawn?q=spawn&v=9.0&api=true.

[31] MarkLogic. 2018. *Developing Modules to Process Content (Content Processing Framework Guide) — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/guide/cpf/modules.

[32] MarkLogic. 2018. *admin:group-add-scheduled-task — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/admin:group-add-scheduled-task.

[33] MarkLogic. 2018. *xdmp:set — MarkLogic 9 Product Documentation*. MarkLogic. https://docs.marklogic.com/xdmp:set.

[34] MarkLogic. 2018. *Understanding Transactions in MarkLogic Server (Application Developer's Guide) — MarkLogic 9 Product Documentation*. *Visibility of Updates*. MarkLogic. https://docs.marklogic.com/guide/app-dev/transactions#id_85012.

[35] James Wright. 2016-02-13. *XML Prague 2016 Conference Proceedings*. 1. 978-80-906259-0-7. *Promises and Parallel XQuery Execution*. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#page=151.

[36] James Wright. *xq-promise*. 2016-04-29. Git Hub. https://github.com/james-jw/xq-promise.

[37] Conway Melvin. 1963. *A Multiprocessor System Design*. ACM. 10.1145/1463822.1463838. *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. 139-146.

[38] 2018-11-15T06:49:39Z. *Promise | MDN. Syntax.* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#Syntax.

[39] 2015. 6th Edition. *Standard ECMA-262. ECMAScript® 2015 Language Specification.* Ecma International. http://www.ecma-international.org/ecma-262/6.0/#sec-promise-executor.

[40] Carl Hewitt. Peter Bishop. Richard Steiger. 1973. *A Universal Modular ACTOR Formalism for Artificial Intelligence*. *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. 235-245. Morgan Kaufmann Publishers Inc.. http://dl.acm.org/citation.cfm?id=1624775.1624804.

[41] Joe Armstrong. Ericsson AB. 2007. *A History of Erlang*. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. 6-1--6-26. ACM. 978-1-59593-766-7. 10.1145/1238844.1238850.

[42] Lightbend, Inc.. *Akka Documentation*. *Actors*. 2018-12-07T11:55:00Z. https://doc.akka.io/docs/akka/2.5.19/actors.html.

[43] 2019-01-17T21:11:00Z. *Actor model*. *Actor libraries and frameworks.* Wikipedia. https://en.wikipedia.org/wiki/Actor_model#Actor_libraries_and_frameworks.

[44] Anders Hejlsberg. Microsoft. 2010-10-28T10:13:00Z. Channel 9. *Introducing Async – Simplifying Asynchronous Programming*. https://channel9.msdn.com/Blogs/Charles/Anders-Hejlsberg-Introducing-Async.

[45] Don Syme. Microsoft Research. 2007-10-10. *Introducing F# Asynchronous Workflows*. https://blogs.msdn.microsoft.com/dsyme/2007/10/10/introducing-f-asynchronous-workflows/.

[46] Simon Marlow. 2012. *async-2.2.1: Run IO operations asynchronously and wait for their results*. *Control.Concurrent.Async.* Hackage. http://hackage.haskell.org/package/async/docs/Control-Concurrent-Async.html.

[47] Mostafa Gaafar. 2017-03-26. *6 Reasons Why JavaScript's Async/Await Blows Promises Away (Tutorial)*. Hacker Noon. https://hackernoon.com/6-reasons-why-javascripts-async-await-blows-promises-away-tutorial-c7ec10518dd9.

[48] Ilya Kantor. 2019. *Promises, async/await*. *Async/await.* JavaScript.info. https://javascript.info/async-await.

[49] Melvin Conway. 1963-07. *Design of a Separable Transition-diagram Compiler*. *ACM Communications*. 6. 396-408. 10.1145/366663.366704. ACM.

[50] Unity Technologies. 2018. *Unity - Manual: Coroutines*. https://docs.unity3d.com/Manual/Coroutines.html.

[51] Harold Coopper. 2012-12. *Coroutine Event Loops in Javascript*. https://x.st/javascript-coroutines/.

[52] Kotlin. 2018-12-06. *Kotlin Documentation. Shared mutable state and concurrency.* GitHub. https://github.com/Kotlin/kotlinx.coroutines/blob/1.1.1/docs/shared-mutable-state-and-concurrency.md.

[53] Domenic Denicola. 2012-10-14. *You're Missing the Point of Promises*. https://blog.domenic.me/youre-missing-the-point-of-promises/.

[54] Simon Peyton Jones. Philip Wadler. 1992. 1993-01. *Imperative Functional Programming*. ACM. *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. 71-84. 0-89791-560-7. 10.1145/158511.158524. https://www.microsoft.com/en-us/research/wp-content/uploads/1993/01/imperative.pdf.

[55] Paul Hudak. John Hughes. Simon Peyton Jones. Philip Wadler. 2007-04-16. *A History of Haskell: Being Lazy with Class*. *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. 12-1--12-55. 978-1-59593-766-7. 10.1145/1238844.1238856. ACM. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf.

[56] The University of Glasgow. 2010. *base-4.12.0.0: Basic libraries. Control.Concurrent.* Hackage. http://hackage.haskell.org/package/base/docs/Control-Concurrent.html#v:forkIO.

[57] John A De Goes. 2017-09-16. *There Can Be Only One...IO Monad*. http://degoes.net/articles/only-one-io.

[58] Alexandru Nedelcu. 2018-11-09. *Task - Monix. Documentation.* GitHub. https://monix.io/docs/3x/eval/task.html.

[59] Viktor Klang. Lightbend, Inc.. 2017-12-19. *Reactive Streams*. GitHub. http://www.reactive-streams.org/.

[60] Mozilla. 2018-09-23T04:04:54Z. *JavaScript - Concurrency model and Event Loop*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[61] Adam Retter. *xq-promise Terminology vs. JavaScript/jQuery*. 2018-11-30. GitHub. https://github.com/james-jw/xq-promise/issues/19.

[62] Adam Retter. 2019-01-13. *Haskell I/O and XPath*. https://blog.adamretter.org.uk/haskell-io-and-xpath/.

[63] Giorgio Ghelli. Christopher Ré. Jérôme Siméon. 2006. *XQuery!: An XML query language with side effects*. *Current Trends in Database Technology -- EDBT 2006*. Springer Berlin Heidelberg. 178-191. 978-3-540-46790-8.

[64] Saxonica. 2018-12-06. *Saxon Documentation. Tuple types.* Saxonica. http://www.saxonica.com/documentation/index.html#!extensions/syntax-extensions/tuple-types.