Task Abstraction for XPDLs

XML Prague 2019

Debbie Lockett <debbie@saxonica.com>

Adam Retter <adam@evolvedbinary.com>

Introduction

Be careful what you wish for...

In this talk...

- Background to problem / What we set out to solve!
- Side effects and concurrency
- Introduction to EXPath Task module - define functions and how to use them
- Code examples and demos using Tasks
- Conclusions and future work

A problem

ixsl:schedule-action is inflexible and exhibits side effects

```
<xsl:template name="send-request">
   <xsl:variable name="request" select="
   map{
      'method': 'POST',
      'href': 'http://localhost:19757/mywebapp/receiveXML',
      'body': $body,
      'media-type': 'application/xml'
      } "/>

   <ixsl:schedule-action http-request="$request">
      <xsl:call-template name="handle-response"/>
   </ixsl:schedule-action>

</xsl:template>

<xsl:template name="handle-response">
   <xsl:context-item as="map(*)" use="required"/>
   <xsl:for-each select="?body">
      <xsl:call-template name="process-response-body"/>
   </xsl:for-each>
</xsl:template>
```

Conclusion: **We need a better way to compute tasks**

A side effect is where external state is mutated

e.g. writing to a file or accessing a web page

Concurrency is where more than one thing appears to happen simultaneously

e.g. printing a document while running some queries

Tea break

or time for some baking...

Baking a cake

Suppose we have a recipe for baking a cake...

Obviously some steps need to be carried out in the order specified.

Bake the cake before icing it.

Baking a cake

Meanwhile it may be OK to do some steps **concurrently**.

Weigh the flour at the same time as cracking the eggs into a bowl.

Care must be taken when reordering to avoid problems from **side effects.**

If you get the cake decorations out too early, there's a risk they won't be there when you get to the decorating stage...

Baking a cake

The recipe will not say *exactly* how long to bake for (only something like "put in the oven for 20 minutes, or until cooked"). While the cake is in the oven, rather than just waiting, the baker is able to get on with other steps. The task is **asynchronous**.

Put the cake in the oven - only take it out and proceed with the subsequent instructions, when it's done.

Meanwhile start preparing the icing, or have a cup of tea.

Task-based cake recipe

Replace standard recipe of sequential steps:

- Wrap each instruction as a **task**
- Explicitly describe how these are composed
- Explicitly allow concurrent or asynchronous processing

The recipe becomes something more like a flow diagram...

The baker now has explicit information about opportunities for reordering and/or concurrency.

EXPath Tasks

It's not all about cakes

General problem

- Safely manage side effects in XPDLs

Recall: by definition side effects are NOT allowed in pure functional languages!

- Permit parallel or concurrent operation

XPDL Existing Solutions

- Side effects
  - Frameworks: XQuery Update - PUL, xq-promise
  - Processors: MarkLogic, BaseX, eXist-db, Saxon-EE/CE

- Concurrency
  - Frameworks: xq-promise
  - Processors: MarkLogic, BaseX, eXist-db, Saxon-EE/JS

Non-XPDL Solutions

- Actors
- Async/Wait
- Co-routines
- Haskell IO Monad
- Promises and Futures
- Reactive Streams

Tasks

Aim: provide a way to safely manage side effects and concurrent execution in XPDLs.

A **task** is an object which encapsulates an action (which may have side effects); which could be *lifted* to be **asynchronous**.

Functions are provided to:

- create and compose tasks
- create and use asynchronous tasks
- manage errors
- execute a task (chain)

How Tasks work

- A Task is a state transformation function
  - Performs your action upon a special **RealWorld** object.
  - Is a pure-function and is " *safe*"!
- When a task chain is actually executed, the *RealWorld* is passed through the chain; which enforces the correct execution order.
- For convenience we represent a Task as XDM map (encapsulating an action):

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

How Tasks work

For any **$task**, the action is stored in the **"apply"** entry of the map:

$task?apply is a function of type:

```
function(element(adt:realworld)) as item()+
```

This function always returns a pair:

```
(element(adt:realworld), $action-result)
```

When tasks are composed, the 2nd task always takes the **RealWorld** from the result of the 1st task; and uses its **$action-result** as needed.

Creating Tasks

- **task:value($v)** - the task's action is to return a value

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

- **task:of($f)** - the task's action is to execute the function and return its result

```
task:of(function() {'hello world'})
```

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

- **task:error($code, $description, $error-object)** - the task's action is to raise an error

```
task:error(
    xs:QName("local:oops"),
    "something went wrong", ())
```

Composing Tasks

Different functions are available for composing tasks (the result of each function is a task):

- **task:bind($task, $binder)** - provide a binder function which creates a new *task* from an existing task's value
- **task:then($task, $next)** - compose two tasks, discarding first task's value
- **task:fmap($task, $mapper)** - provide a mapper function which creates a new *value* from an existing task's value
- **task:sequence($tasks)** - create a new task from the sequential application of one or more tasks

Executing Tasks

Function for executing a task (chain):

- **task:RUN-UNSAFE($task)** - executes a task chain, and returns the result.

WARNING: This function is **inherently unsafe**, as it causes any side effects within the task chain to be actualised. It should only be invoked once in any application; at the end.

Asynchronous tasks

Function to construct an asynchronous task:

- **task:async($task)** - constructs an asynchronous task from an existing task

When an **asynchronous task** is executed, it returns an **Async** - a function which represents the asynchronous process, not the result of that process.

An **Async** is an "*abstract type*":

```
function(element(adt:scheduler)) as ~A
```

...~A is the type of the result of the asynchronous process

Functions upon Async

- **task:wait($async)** - extracts the value of an Async and returns a task of the value; possibly by blocking
- **task:wait-all($asyncs)**
- **task:cancel($async)** - attempt to cancel the asynchronous process
- **task:cancel-all($asyncs)**

Alternative Map Syntax

Using maps for tasks means we can provide alternative imperative-like fluent syntax for these functions:

Function based syntax	**Imperative-like syntax**
task:bind($task, $binder)	$task ? bind($binder)
task:then($task, $next)	$task ? then($next)
task:fmap($task, $mapper)	$task ? fmap($mapper)
task:sequence($tasks)	$task1 ? sequence(tail($tasks))
task:async($task)	$task ? async()
task:RUN-UNSAFE($task)	$task ? RUN-UNSAFE()

Examples & Demos

Let's see some code!

Demos using EXPath Task implementation in XQuery or XSLT

Cake baking

Starting to write a cake recipe using Tasks

```
task:of(function() { local:make-cake#0})
      ? fmap(local:bake#1)
      ? fmap(local:decorate-cake#1)
      ? async()
      ? bind(task:wait#1)
      ? RUN-UNSAFE()
```

Asynchronous HTTP 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>
```

Conclusions

Was it all worth it?

Benefits of EXPath Tasks

The EXPath Task module meets our initial requirements:

- Allows developers to safely encapsulate side-effecting functions in XPDLs so that at evaluation time they appear as pure functions, and enforce the expected order of execution
- Allows concurrent programming to be explicitly described; implementable on systems offering preemptive or cooperative multitasking

Using Tasks

How well can use of the Task module be incorporated into IXSL stylesheets for Saxon-JS applications?

- Still at an early stage of evaluation
- Likely to require significant application restructure, but benefits should make this worthwhile
- May require new IXSL extensions to be able to code certain mechanisms nicely (e.g. providing an abort button for an asynchronous HTTP request)

Future Work

- Develop implementations further (get asynchronous actions working!)
- Community feedback
- Task Module additions
- EXPath Spec 1.0?

Implementations

- XQuery
  https://github/adamretter/task.xq
- XSLT
  https://github/saxonica/expath-task-xslt
- Java (for XQuery in eXist-db)
  https://github.com/eXist-db/exist/tree/expath-task-module-4.x.x/ex- tensions/expath/src/org/expath/task
- JavaScript (for XSLT in Saxon-JS)

Thanks for listening

Any questions?