# F# Async in Dev11

List of features we might consider adding for dev11:

1. Async state machines
2. AsyncResultCell – the implementation of a Future for asyncs
3. Task interop
4. IAsyncDisposable
5. ThreadAffinityContext
6. asyncSeq
7. Enabling async hops across domains
8. Making MailboxProcessor MarshalByRef

## Async State Machines

See allocation-free-async-workflows.docx

## AsyncResultCell

We ship AsyncResultCell in PowerPack today, and we have an internal implementation of the same in FSharp.Core. The F# async version of the Future proved to be extremely useful in many scenarios – we must make it available to F# programmers out of the box.

## Task Interop

TBD

## IAsyncDisposable

So in our today’s implementation of asynchronicity in a language service, we use this pattern:

    member info.GetDeclarations((line,colAtEndOfNames),lineStr,names:NamesWithResidue,tokenTag:int) =

        async {

            do! switchToReactorThread

            return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

        }

And at the call site:

async {

    let! decls = req.ResultScope.GetDeclarations(req.View, req.Line, req.Col, req.TokenInfo, reason)

    do! Async.SwitchToContext UIThread.TheSynchronizationContext

    …

}

This is an example of code that is correct (well, there is a bit of exception handling issue, but let’s ignore that for the sake of this discussion), and *in the given circumstances is actually an “optimal” one*, but is an architecture nightmare. There are two reasons to that, one has to do with suboptimal story for thread affinity in F# asyncs, and one with the design disaster that is SynchronizationContext.Current. Let’s go over these in order and examine the solutions.

What the code above wants to do (and it is in fact quite typical scenario) is:

* Execute “DeclarationSet(scope.GetDeclarations(),…)” asynchronously on *reactor* thread.
* Get back on the UI thread

The way the code is written above, the continuation of the async that GetDeclarations returns *will continue to run on  reactor thread*. That is why we need Async.SwitchToContext at the call site. Obviously, GetDeclarations shouldn’t let the continuation run on reactor thread – it wants the continuation to return to the “thread”, or “context” where it came from. The abstraction we use for that today is SynchronizationContext, so the code you will want to write looks like this:

        async {

            let sc = System.Threading.SynchronizationContext.Current

            try

                do! switchToReactorThread

                return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

            finally

                do! Async.SwitchToContext sc

        }

Which has two issues:

1. It is really ugly and does not convey the intent well
2. You cannot actually write that today, because “finally” blocks do not let you bind inside of them!

Thanks to Tomas, we have a suggestion of how to deal with this: “use!” and IAsyncDisposable to the rescue:

        async {

            use! \_ = Async.ReactorThread

          return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

        }

where Async.ReactorThread is an Async<IAsyncDisposable> which captures the sync context, schedules the continuation onto reactor thread and in Dispose async schedules continuation back onto the original sync context.

Our general guidance says that all async APIs should return back to SynchronizationContext they have been called on. This style of programming gives F# programmer a really nice tool for following that guidance.

The callsite will also look rather nice:

async {

       use! \_ = Async.SynchronizationContext UIThread.TheSynchronizationContext

let! decls = req.ResultScope.GetDeclarations(req.View, req.Line, req.Col, req.TokenInfo, reason)

…

}

This code *advertises* that the async in question should run on UI thread.

In general, with the “use! Async.*ThreadAffinity*” the async block can *declare* the thread affinity it requires – it looks lovely!

## ThreadAffinityContext

All of the above is lovely, but unfortunately does not quite work in VS – because SynchronizationContext is such a disaster:

1. As we have learned during dev10 cycle, in VS UI thread does not necessarily has SynchronizationContext.Current set. Therefore even if all other API calls play nicely, we cannot guarantee that they return UI thread after all, so we will still need to litter our code with “do! Async.SwitchToContext UIThread.TheSynchronizationContext” after every “let!” on UI thread – yuck!
2. When GetDeclarations runs its async on reactor thread, it wants every “let!” inside that async to return to reactor thread (that is what “use! Async.ReactorThread” should imply). This means installing SynchronizationContext.Current on reactor thread – but if I do that, I am sure to get an inquisitive e-mail from Doug Hodges or Paul Harrington or such like . In general, setting the current synchronization context is a security critical operation and shouldn’t be required for enforcing async thread affinity.

Generally, the big problem with SynchronizationContext.Current is that it is a universally accessible global and breaks encapsulation (as is universally accessible globals’ habit).

The proposed solution here is to make an equivalent of SynchronizationContext (let’s call it ThreadAffinityContext) a part of Async monadic state (aka AsyncParams).

We will have operations like:

GetThreadAffinity : Async<ThreadAffinityContext>

SetThreadAffinity : ThreadAffinityContext -> Async<unit>

ThreadAffinity : ThreadAffinityContext -> Async<IAsyncDisposable>

The current ThreadAffinityContext of the async monad will be used in places where we today use SynchronizationContext.Current.Post.

For backwards compatibility, the default ThreadAffinityContext asyncs run on will delegate to SynchronizationContext.Current.

We have precedents for the this kind of thing in other async frameworks –TPL and Rx have TaskScheduler and IScheduler abstractions respectively for this kind of stuff. We should check with them for possible alignment here. There are also alignment issues with C#/VB feature that we will need to work out.

## AsyncSeq

Tomas, that’s a wonderful contribution! Indeed, it feels like this may be heading towards something that is worth a paper (all together?).

I’m still mulling over your type definition below. It is certainly beautiful and natural. I would instinctively convert it to the following (following F# lazy lists, and Wadler/Taha’s paper “[How to add laziness to a strict language, without even being odd](http://citeseer.ist.psu.edu/old/725251.html)”, which applies to pretty much an computational modality, not just laziness)

type AsyncSeq<'T> = Async<AsyncSeqCell<'T>>

and AsyncSeqCell<'T> = Nil | Cons of 'T \* AsyncSeq<'T>

with typical operations like these (so simple!!)

let rec map (f: 'T -> 'U) (x: AsyncSeq<'T>) : AsyncSeq<'U> =

    async { let! v = x

            match v with

            | Nil -> return Nil

            | Cons (h,t) -> return Cons(f h, map f t) }

let rec mapAsync (p: 'T -> Async<'U>) (x: AsyncSeq<'T>) : AsyncSeq<'U> =

    async { let! v = x

            match v with

            | Nil -> return Nil

            | Cons (h,t) -> let! h2 = p h in return Cons(h2, mapAsync p t) }

I attach the builder implementation that follows for this type. Everything runs through very smoothly. The tryFinally and tryWith are pretty interesting (I have not written anything like them before), and thus not yet fully tested, though I think they are right, and they pass all the tests I had laid out before.

Now, the question is, **what does this actually do**?

Well, the answer is that it **time is no longer in the picture ,** at leastuntil you start connecting to time-volatileevent streams.  That is, **these are not event streams**. They are simply sequences whose “head/tail” cell is generated asynchronously.

So basically, in every case you get the same answer as a synchronous sequence, e.g. this:

              asyncSeq { for x in asyncSeq { yield 1;

                                             do! Async.Sleep 100

                                             yield 2 } do

                            yield (x,'a')

                            do! Async.Sleep 10

                            yield (x,'b') }

gives

 [(1,'a'); (1, 'b'); (2,'a'); (2,'b') ]

This is regardless of the time delays involved in either sequence – as I said, time is not in the picture. We do these asynchronous waits@

input  Cons(1,tail1)

sub-list-for-1  Cons((1,’a’), tail2)

tail2  <pause-10>  Cons((1,’b’), tail3 )

tail3  Nil

tail1  <pause-100>  Cons(2,tail4)

sub-list-for-2  Cons((2,’a’), tail5)

tail5  <pause-10>  Cons((2,’b’), tail6 )

tail6  Nil

I also like the fact that the relationship to events and observables becomes very clear, e.g. if you look at these, then you see straight away that the asynchronous sequence is continually reconnecting to the event stream and/or observable.

    module AsyncSeq =

        // This may skip event observations if upstream processing takes too long

        let rec ofEvent (ev: IEvent<'T>) : AsyncSeq<'T> =

            asyncSeq { let! v = Async.AwaitEvent ev

                       yield v

                       yield! ofEvent ev }

        // This continually reconnects, and may miss observations if upstream processing takes too long

        let rec ofObservable (ev: System.IObservable<'T>) : AsyncSeq<'T> =

            asyncSeq { let! obs = Async.AwaitObservable ev

                       yield obs

                       yield! ofObservable ev }

And presumably a zip combinator would allow a “merge” story if we wanted it (oh, but “zip” would involve a parallel wait on one of two asyncs – tricky without introducing multi-threading – I hear Dmitry asking for interleaved asyncs on a single thread )

Anyway, all very intriguing, and overall a much cleaner story.

Don

**From:** Tomas Petricek [mailto:tomas@tomasp.net]   
**Sent:** 17 February 2010 21:06  
**To:** Dmitry Lomov; Don Syme; Brian McNamara; Matthew Podwysocki  
**Subject:** RE: AsyncSeq (= Observable) puzzles

I was following the discussion and I can only agree that this is *really* tricky.

**Representing AsyncSeq in F#**

Maybe the problem is that a natural representation of asynchronous sequence in F# is not IObservable, but something like this:

type AsyncSeq<'T> = Nil | Cons of Async<'T \* AsyncSeq<'T>

...the difference would be that this representation is delayed, so when you “bind” on the value of Cons, it evaluates (asynchronously) only one step of the sequence. I believe that this type would naturally work with “async” in the same way F# list works in the context of normal code (you can use it for recursive programming with pattern matching (using “match!” of course!) and the “for” loop would evaluate the asynchronous sequence lazily, so it could behave as the version that doesn’t miss any values, but without any caching.

I haven’t really tried this, so I may be completely wrong. Anyway, if I’m correct, then we could use this data type for asynchronous sequences in F# and then define various conversion functions to get from this to IObservable and back (when getting from IObservable it could use caching or it could miss values). Anyway, this is probably purely theoretical, because adding even more types for asynchronous/reactive/parallel programming would be too confusing.

**AsyncSeq for IObservable**

Back to possible designs of “asyncSeq” computation builder for IObservable:

* I think that it should be resumption (one-shot continuation) and there should be only one program counter  
  (F# asynchronous workflows work this way and I think this an important aspect that makes them intuitive and easy to use)
* I was going to suggest that we add something like AsyncEnumerator (which allows explicit access to elements and implements caching), but it seems that you already have that (AsyncObserver), which is great! The similarity between GetEnumerator and AsyncObserver is probably a great way to explain the programming model.
* For resumption, the only question is whether “for” should automatically cache the values or not. I think losing values when using AwaitEvent is a good thing, but I’m not really sure about “for”.

Are there any other code samples that we would expect to be equivalent? For example this one:

|  |  |
| --- | --- |
| asyncSeq {  **for** *v* **in** *aseq* **do**  *<body>* } | **let** ao = AsyncObserver(*aseq*)  **let rec** loop() = asyncSeq {  **let!** next =ao.Next  **match** next **with**    | Some(v) -> *<body>*; return! loop()  | None -> () |

This would suggest that “for” should be using caching. So, I my vote would probably be to use caching. Another practical question is whether the user can easily choose between the two. For example, can he write **for** v **in** aseq |> AsyncSeq.forgetful **do** if we use the cached option (this seems to be tricky)? If we choose the non-caching version, can the user write **for** v **in** aseq |> AsyncSeq.forgetful **do** (this looks more plausible). Anyway, this may be another thing to consider.

**Off-topic: Asynchronous waiting in while**

And here is one more off-topic note – in the equality above, the recursive version on the right hand side was surprisingly difficult to write! I was expecting something like (with “while!” which does asynchronous waiting when evaluating the condition – technically speaking it takes “unit -> Async<bool>” instead of “unit -> bool”):

**let** ao = *aseq*.GetAsyncEnumerator()

**while!** ao.MoveNext() do

**let** v =ao.Current

*<body>*

Thanks for including me in this discussion!

Tomas

## Making MailboxProcessor MarshalByRef

## Enabling async hops across domains