Extensions

An easy way to put what we've learned thus far to use is in extending existing computation expressions. We one way do to this in the second exercise where we created the `ChoiceBuilder` by inheriting from `OptionBuilder`. Inheritance, however, is not always an option. In this exercise, we'll look at two more ways to extend existing builders, both leveraging type extensions:

1. Method overloads
2. `CustomOperations`

Fortunately, CEs can pick up any type extensions to builder types, so you can define these in other modules, libraries, etc. This means you could define different extensions for different use cases and only open the extensions you want in a given context.

In this exercise, we'll extend both the `Microsoft.FSharp.Control.AsyncBuilder` and `OptionBuilder` types with new functionality.

1. Create a new file, `Extensions.fs`.
2. Add the file to your `.fsproj` with `<Compile Include="Extensions.fs" />` just below `ExceptionsLoopsDisposal.fs`.
3. Add the following lines to the `Extensions.fs` file:

```
module Extensions

open Expecto
open Options

[<Tests>]
let tests =
    testList "extensions" [
    ]
```

Type Extensions (Optional)

This section serves as a primer for those unfamiliar with type extensions.

Intrinsic Type Extensions

Intrinsic type extensions are defined next to the type within the same code file, namespace, etc. Here's an example:

```
type USAddress =
    {
        Recipient : string
        Street1 : string
        Street2 : string
        City : string
```

```
        State : string
        Zip : string
    }

type USAddress with
    member this.Format() =
        let sb = System.Text.StringBuilder()
        sb.AppendLine(this.Recipient)
          .AppendLine(this.Street1)
          .AppendLine(this.Street2)
          .Append(this.City)
          .Append(", ")
          .Append(this.State)
          .Append(" ")
          .AppendLine(this.Zip)
```

Even though the `Format` method is defined in a type extension, this method will be included in the core definition of the `USAddress` type and may exist alongside in either a `namespace` or `module`. This type of extension is useful when you want to separate fields and behaviors or when you may want to define helper functions that will be referenced in the methods but that should not be part of the `type` definition itself.

Optional Type Extensions

Optional type extensions may be defined outside the original `type` definition and may be applied to types from other libraries. These ***are not*** made a part of the type and ***must*** be declared within a `module`. Here's an example:

```
module Extensions

open Expecto

type Microsoft.FSharp.Control.AsyncBuilder with
    member this.Foo(bar) = sprintf "foo%sbaz" bar

let result = async.Foo("bar")
```

The `async` CE builder now has a `Foo` member you can call. You did not modify the `AsyncBuilder` type directly, but the compiler will respect the member as an optional part of the type.

Method Overloading

A common complaint when working with C# code is having to translate between `Task<'T>` (and `Task`) and `Async<'T>` with `Async.AwaitTask`. Wouldn't it be nice if the `async { let! res = task }` could just bind to a `Task<'T>` or `Task`?

Despite the specified signature of `Bind` from the docs, `M<'T> * ('T -> M<'U>) -> M<'U>`, we can achieve this feature using method overloading. It turns out that `M` is not required to remain the same type all over.

Add the following test case:

```
        test "async can bind Task<'T>" {
            let expected = 0
            let task = System.Threading.Tasks.Task.FromResult expected
            let actual =
                async {
                    let! res = task
                    return res
                }
                |> Async.RunSynchronously
            Expect.equal actual expected "Expected async to bind Task<'T>"
        }
```

We need to teach `AsyncBuilder` how to handle `Task<'T>`, which is easily implemented with `Async.AwaitTask`. Add the following type extension above your tests:

```
type Microsoft.FSharp.Control.AsyncBuilder with
    member this.Bind(task, f) =
        this.Bind(Async.AwaitTask task, f)
```

Once you add this implementation, you should be able to confirm that the F# compiler correctly infers the type of your `Bind` member as:

```
System.Threading.Tasks.Task<'a> * ('a -> Async<'b>) -> Async<'b>
```

In addition, the red squiggle in your test case indicating a compiler error should have disappeared.

> **Observation:** this once again highlights the flexibility of the CE mechanism to adapt beyond the rigid type constraints found in the documentation.

Run your tests with `dotnet test` to see your tests pass.

Let's add another test case:

```
        test "async can bind Task" {
            let expected = 0
            let task =
                System.Threading.Tasks.Task.FromResult ()
                :> System.Threading.Tasks.Task
            let actual =
                async {
                    do! task
                }
                |> Async.RunSynchronously
            Expect.equal actual expected "Expected async to bind Task"
        }
```

Unfortunately, our overload does not handle the non-generic `Task`. That's okay, we can create another overload to handle that case:

```
    member this.Bind(task:System.Threading.Tasks.Task, f) =
        this.Bind(Async.AwaitTask task, f)
```

More recent versions of F# have an `Async.AwaitTask` that can await non-generic tasks. However, if you are using an older version of FSharp.Core, you may need a more involved implementation:

```
    member this.Bind(task:System.Threading.Tasks.Task, f) =
        // One of many possible implementations ...
        this.Bind(Async.AwaitTask(task.ContinueWith(System.Func<_,_>(fun _
-> ()))), f)
```

Similar extensions may be made to our `OptionBuilder` from the first exercise. Add the following tests:

```
        test "maybe can bind Nullable<'T>" {
            let expected = 1
            let nullable = System.Nullable(1)
            let actual =
                maybe {
                    let! x = nullable
                    return x
                }
            Expect.equal actual (Some expected) "Expected a Nullable 1 to
return Some 1"
        }

        test "maybe can bind a null object" {
            let expected : string = null
            let actual =
                maybe {
                    let! x = expected
                    return x
                }
            Expect.equal actual None "Expected a null object to return
None"
        }
```

Add overloads of `Bind` in our `Extensions` module to satisfy the tests above. Once complete, you should be able to run `dotnet test` and see all tests pass successfully.

CustomOperations

Another way to extend a computation expression is with a `CustomOperations`. We'll see these in greater detail in the next section.

`CustomOperation`s were introduced in F# 3.0 alongside Type Providers and therefore missed the fanfare they very richly deserve. These let you add custom syntax into your computation expressions, giving you the ability to add new "keywords" into your programs.

In this exercise, we'll look at adding a proposed language extension into the `async {}` computation today.

`async.Bind` awaits an async computation and provides the result to its callback function. This allows you to chain dependent `async` computations together, e.g.

```
let example =
    async {
        let! a = first ()
        let! b = second a
        let! c = third c
        return c
    }
```

What if you don't have dependent `async` computations and would prefer them to run concurrently? With the current `async` computation, you could do the following, using the new `match!` in F# 4.5:

```
let example =
    async {
        match! Async.Parallel [|async.Return(1); async.Return(2);
async.Return(3)|] with
        | [|first; second; third|] ->
            return first + second + third
        | _ ->
            failwith "Impossible scenario"
            return -1
    }
```

While the above works, it requires dealing with pattern matching the result, even though the number of items in the array is known. Wouldn't it be nice to have something a bit more like `let!` but without requiring sequential processing?

A language suggestion proposes the addition of `let! ... and! ...` syntax that would allow exactly this. Of course, this syntax is not currently available within computation expressions ... or is it?

```
        test "concurrent async execution" {
            let expected = 3
            let parallel =
                async {
                    let! a = async.Return(1)
                    and! b = async.Return(2)
                    return a + b
                }
            let actual = Async.RunSynchronously parallel
```

```
            Expect.equal actual expected "Expected actual to equal 3"
        }
```

Add the above test, and the compiler will greet you with red squiggles indicating, "Identifiers followed by '!' are reserved for future use." What about the following?

```
        test "concurrent async execution" {
            let expected = 3
            let parallel =
                async {
                    let! a = async.Return(1)
                    ``and!`` b = async.Return(2)
                    return a + b
                }
            let actual = Async.RunSynchronously parallel
            Expect.equal actual expected "Expected actual to equal 3"
        }
```

This produces the error, "The value or constructor 'and!' is not defined." Progress! Now we need to support this new syntax with a `CustomOperation`. Just as we did above, we need to add a type extension:

```
type Microsoft.FSharp.Control.AsyncBuilder with
    [<CustomOperation("and!")>]
    member this.Merge(x, y, resultSelector) =
        async {
            let! [|x';y'|] = Async.Parallel [|x;y|]
            return resultSelector x' y'
        }
```

This is the simplest way to construct a custom operation, but the compiler informs us this is incorrect:

```
'and!' is not used correctly. This is a custom operation in this query or
computation expression.
```

`CustomOperationAttribute` has a number of additional settings you can use:

Name	Description
`AllowIntoPattern : bool with get, set`	Indicates if the custom operation supports the use of into immediately after the use of the operation in a query or other computation expression to consume the results of the operation.

Name	Description
`IsLikeGroupJoin : bool with get, set`	Indicates if the custom operation is an operation similar to a group join in a sequence computation, in that it supports two inputs and a correlation constraint, and generates a group.
`IsLikeJoin : bool with get, set`	Indicates if the custom operation is an operation similar to a join in a sequence computation, in that it supports two inputs and a correlation constraint.
`IsLikeZip : bool with get, set`	Indicates if the custom operation is an operation similar to a zip in a sequence computation, in that it supports two inputs.
`JoinConditionWord : string`	Indicates the name used for the "on" part of the custom query operator for join-like operators.
`MaintainsVariableSpace : bool with get, set`	Indicates if the custom operation maintains the variable space of the query of computation expression.
`MaintainsVariableSpaceUsingBind : bool with get, set`	Indicates if the custom operation maintains the variable space of the query of computation expression through the use of a bind operation.

We will explore these more in the next exercise. For now, we will focus on `IsLikeZip`. This allows us to create a custom operation that will take two things and pair them up, which is essentially what we want to do to run things concurrently.

We also need to add an attribute to our `resultSelector`. This is the callback parameter, or the continuation that will be called with our zipped `async` computations. We need to help the compiler by specifying that this is a `[<ProjectionParameter>]`.

Our revised extension looks as follows:

```
type Microsoft.FSharp.Control.AsyncBuilder with
    [<CustomOperation("and!", IsLikeZip = true)>]
    member this.Merge(x, y, [<ProjectionParameter>] resultSelector) =
        async {
            let! [|x';y'|] = Async.Parallel [|x;y|]
            return resultSelector x' y'
        }
```

The compiler now reports,

```
The type 'Async<int * int>' is not compatible with the type 'seq<'a>'
```

This is not immediately helpful, but the type `seq<'a>` and the concept of `zip` provide some context clues. This `IsZipLike` behavior expects a sequence as an input, which implies we need a `For` member:

```
    member this.For(m, f) = this.Bind(m, f)
```

We'll also need to change our syntax a bit:

```
        test "concurrent async execution" {
            let expected = 3
            let parallel =
                async {
                    for a in async.Return(1) do
                    ``and!`` b in async.Return(2)
                    return a + b
                }
            let actual = Async.RunSynchronously parallel
            Expect.equal actual expected "Expected actual to equal 3"
        }
```

This does not exactly match our desired syntax, but it does provide an immediate solution. Our test does not really highlight any benefit of this change. Let's create a comparison between the two syntaxes to verify that our solution works.

```
        test "concurrent runs concurrently, not sequentially" {
            let task1 =
                async {
                    do! Async.Sleep(1000)
                    return 1
                }
            let task2 =
                async {
                    do! Async.Sleep(1000)
                    return 2
                }
            let sequentialStopwatch = System.Diagnostics.Stopwatch()
            let sequential =
                async {
                    sequentialStopwatch.Start()
                    let! a = task1
                    let! b = task2
                    sequentialStopwatch.Stop()
                    return a + b
                }
                |> Async.RunSynchronously
            let parallelStopwatch = System.Diagnostics.Stopwatch()
            let parallel =
                async {
                    parallelStopwatch.Start()
                    for a in task1 do
                    ``and!`` b in task2
                    parallelStopwatch.Stop()
```

```
                    return a + b
                }
                |> Async.RunSynchronously
            printfn "Sequential: %d, Concurrent: %d"
                sequentialStopwatch.ElapsedMilliseconds
                parallelStopwatch.ElapsedMilliseconds
            Expect.equal parallel sequential
                "Expected parallel result to equal sequential result"
            Expect.isLessThan
                parallelStopwatch.ElapsedMilliseconds
                sequentialStopwatch.ElapsedMilliseconds
                "Expected parallel to be less than sequential"
        }
```

When you run `dotnet test`, you should see all tests pass, and you should see an output of the form, `Sequential: 2009, Concurrent: 1000`, though the actual elapsed milliseconds may be different. Note that the concurrent result should be roughly half the time of the sequential result.

> **NOTE:** The above implementation is not the only solution, and it has a downside in that the inputs must have the same type, i.e. `Async<'a> * Async<'b> * ('a -> 'a -> 'b) -> Async<'b>`. It's possible to implement this to take different input types, e.g. `Async<'a> * Async<'b> * ('a -> 'b -> 'c) -> Async<'c>`. Tip: try using `Async.StartAsChild` or `Async.StartAsTask` rather than `Async.Parallel`.

As a further exercise, try implementing an improved version of `Merge` that accepts different input types as described above, **or** look for another way to achieve the same result with a different solution, e.g. try overloading `Combine` as we did in previous exercises.

Review

In this section, we looked at two ways to extend computations:

1. Method overloading
2. `CustomOperation`s

This is a pragmatic (and fun!) way to work with computation expressions in real projects.

We only scratched the surface of `CustomOperation`s. In the next section, we'll look at these in more detail in the context for which they were created: Query Expressions.