Embedded Domain Specific Languages

Thus far, we've looked at the building blocks of computation and query expressions mostly from the perspective of the purpose for which they were created. Now we are going to look at examples that highlight a capability they enable, specifically embedded domain specific languages, or eDSLs.

We observed in the Extensions section that `CustomOperation`s could be used to create any new syntax you desire. In `Queries` we looked into how each of the `CustomOperation` attribute options could be applied to build query operators. As these are the intended uses, they are easy to review and teach. EDSLs, are specific to each individual use case, so the best way to learn from them is to review their source code.

We'll look at the following open source projects and how they leverage `CustomOperation`s to achieve their objectives:

1. NuGet
2. IL
3. Saturn
4. Freya

We'll also take a look at how Freya achieves its `CustomOperation` overloading, which is not supposed to be possible, as we saw at the end of the last section.

NuGet

These NuGet builders were defined in the article Embedded Domain Specific Languages in F# Using Custom Operations, describing the potential for `CustomOperation`s to be used to create eDSLs.

- `PackageDefinitionBuilder`
- `NuGetDefinitionBuilder`

> **Observation:** these are *very* simple implementations with *no* standard CE methods and no use of `CustomOperation` attribute parameters.

IL

Generating IL at runtime is something of an artform, but rarely do .NET developers write literal IL to do so. The following projects allow F# developers to do almost exactly that, embedded ***within*** their F# programs.

- ILBuilder uses a Type Provider ***within*** the builder definition
- ILBuilder leverages Type Providers ***when using*** the builder
- LicenseToCIL uses `Reflection.Emit`

Saturn

Saturn is an MVC-style framework built on top of Giraffe's `HttpHandler`, or `HttpContext -> Async<HttpContext option>`. It provides a number of eDSLs for composing applications from various parts, each easily composed alongside standard `HttpHandler`s.

- `ControllerBuilder`

- `ApplicationBuilder`
- `PipelineBuilder`
- `RouterBuilder`

Freya

Freya is another web ***stack*** that provides a functional-first, type-safe approach to building web applications on top of the OWIN spec. Freya provides additional application layers that add types to HTTP, provides routing using URI Templates, and a `webmachine`-style model inspired by the Erlang project, which implements the HTTP state machine described in the HTTP RFCs.

`Freya` = `AsyncState`

`FreyaBuilder`, where `Freya` is a `State -> Async<'a * State>` (for a pre-defined `State` type)

Routers

`UriTemplateRouterBuilder` inherits from `ConfigurationBuilder`

Machines

- `HttpMachineBuilder` inherits from `ConfigurationBuilder`
- `Freya.Machines.Http.Cors` is a library of extensions for `HttpMachineBuilder`

Overloading

One of the most interesting aspects of Freya is how it manages to get around the restriction of overloading `CustomOperation`s. Freya leverages Statically Resolved Type Parameters, or SRTP, to provide an inference mechanism. Here's an example from the `Freya.Machines.Http` library:

```
module Inference =

    [<RequireQualifiedAccess>]
    [<CompilationRepresentation
(CompilationRepresentationFlags.ModuleSuffix)>]
    module DateTime =

        (* Inference *)

        [<RequireQualifiedAccess>]
        module Inference =

            type Defaults =
                | Defaults

                static member DateTime (x: Freya<DateTime>) =
                    Dynamic x

                static member DateTime (x: DateTime) =
                    Static x
            let inline defaults (a: ^a, _: ^b) =
                ((^a or ^b) : (static member DateTime: ^a ->
Value<DateTime>) a)

            let inline infer (x: 'a) =
                defaults (x, Defaults)

        let inline infer v =
            Inference.infer v
```

Going back to our `Extensions` section, would it be possible to support `Task<'T>` in

```
async {
    for x in task1 do
    ``and!`` y in task2
    return x + y
}
```

> **NOTE:** It turns out that you don't *need* this inference mechanism. While `CustomOperation` attributes with the same name may not be used on multiple methods within a class definition, the standard method overloading approach *still* works.

Open your `Extensions.fs` again, and add the following just above your tests:

```
type Microsoft.FSharp.Control.AsyncBuilder with
    member __.Merge(x:Async<'a>, y:System.Threading.Tasks.Task<'b>,
[<ProjectionParameter>] resultSelector) =
        async {
            let x' = Async.StartAsTask x
            do System.Threading.Tasks.Task.WaitAll(x',y)
            return resultSelector x'.Result y.Result
        }
    member __.Merge(x:System.Threading.Tasks.Task<'a>, y:Async<'b>,
[<ProjectionParameter>] resultSelector) =
        async {
            let y' = Async.StartAsTask y
            do System.Threading.Tasks.Task.WaitAll(x,y')
            return resultSelector x.Result y'.Result
        }
    member __.Merge(x:System.Threading.Tasks.Task<'a>,
y:System.Threading.Tasks.Task<'b>, [<ProjectionParameter>] resultSelector)
=
        async {
            do System.Threading.Tasks.Task.WaitAll(x,y)
            return resultSelector x.Result y.Result
        }
```

Add the following test:

```
        test "concurrent task execution within async" {
            let expected = 3
            let concurrent =
                async {
                    for a in System.Threading.Tasks.Task.FromResult(1) do
                    ``and!`` b in
System.Threading.Tasks.Task.FromResult(2)
                    return a + b
                }
            let actual = Async.RunSynchronously concurrent
            Expect.equal actual expected "Expected actual to equal 3"
        }

        test "concurrent async & task execution within async" {
            let expected = 3
            let concurrent =
                async {
                    for a in async.Return(1) do
                    ``and!`` b in
System.Threading.Tasks.Task.FromResult(2)
                    return a + b
                }
            let actual = Async.RunSynchronously concurrent
            Expect.equal actual expected "Expected actual to equal 3"
        }

        test "concurrent task & async execution within async" {
            let expected = 3
            let concurrent =
                async {
                    for a in System.Threading.Tasks.Task.FromResult(1) do
                    ``and!`` b in async.Return(2)
                    return a + b
                }
            let actual = Async.RunSynchronously concurrent
            Expect.equal actual expected "Expected actual to equal 3"
        }
```

Running `dotnet test` should show all tests passing.

Review

In this section, we spent most of our time reviewing other projects that leverage CEs and `CustomOperation`s to create embedded domain specific languages. However, we didn't write a lot of code, as it's not easy to create examples outside a specific use case.

In order to resolve this state of affairs, the remaining time is left to you to create your own computation expression to practice what you've learned. Feel free to use any idea you have, pair up with others, ask for help, etc.

If you don't have an idea of your own, we have two problems prepared that you can work through:

1. Fizz-Buzz checker - create an eDSL that let's you write the expected output of Fizz-Buzz and check the results. This is a play on a classic interview question to turn the tables and turn this into a game, i.e. "Are you smarter than a Fizz-Buzz?"
2. Conditional Probabilities - implement multiple strategies for determining probability using a standard interface, and write expressions that accept the builder interface as a parameter, rather than using a value. For example, one implementation may output the list of all the possibilities, and another can randomly select a possibility at each point. (Proposed by Michael Newton)
3. Generate a computation expression from a Type Provider (an inverse of the post Managing Mutable State With Computational Expressions)