Efficient discriminated unions in C#7

October 28, 2016 C#, .Net edit

Also known as tagged unions, variant records, sum types or even enums (in **Rust enums** or **Swift enums**), they are versatile lightweight types commonly used in functional languages. F# has discriminated unions, but if you take the time to discover what kind of .NET code it generates, you will find that they are stored as classes, hence not particularly optimal.

In this post, I would like to propose an implementation of discriminated unions as valuetypes instead. I will try to cover some of the pitfalls and issues, but unlike my previous posts, I will not provide a full prototype this time (doh!), but it will try to reuse some of my previous adventure.

Why?

Using a managed hierarchy for unions classes is not ideal for many reasons:

- **Size memory cost** of the reference indirection, object header: on a x64 runtime, an union would occupy at least: 8 bytes for the reference, 4 (object header) + 8 (vtbl pointer) + 4 (int id for the union), so in total 24 bytes
- **Bad cache locality** due to the reference indirection
- Puts more **pressure on the GC** as it will have to visit each union instances
- **Inefficient pattern matching**: If not carefully crafted, e.g when using RTTI/type information for example (as F#), the code will be resolved into a bunch of `if type is Union1 { ... }` instead of an efficient switch case. That could be solved by having an enum type field, but you would still pay the cost of the reference indirection.

For example, if we take the following F# example:

```
type Shape =
    | Rectangle of width : float * length : float
    | Circle of radius : float
    | Triangle of width : float * height : float
```

You can check the generated .NET code for it on **this gist**

While a F# pattern matching:

```
module Functions =
 let getShapeSurface shape =
    match shape with
    | Rectangle(w,l) -> w * l
    | Circle(r) -> 2. * r * 3.14159
    | Triangle(w,h) -> w * h / 2.
```

would generate the following .NET code:

```
// TestSharp.Functions
public static double getShapeSurface(Shape shape)
{
    if (shape is Shape.Circle)
    {
        Shape.Circle circle = (Shape.Circle)shape;
        double r = circle._radius;
        return 2.0 * r * 3.14159;
    }
    if (!(shape is Shape.Triangle))
    {
        Shape.Rectangle rectangle = (Shape.Rectangle)shape;
        double w = rectangle._width;
        double i = rectangle._length;
        return w * i;
    }
    Shape.Triangle triangle = (Shape.Triangle)shape;
    double w2 = triangle._width;
    double h = triangle._height;
    return w2 * h / 2.0;
}
```

With discriminated unions as valuetypes, we could more efficiently represents and manipulates them.

Syntax in C#

I will introduce here just a simple syntax for the main purpose of describing after how it will be projected back to the implementation. So syntax is less critical here, though I will try to avoid introducing a new keyword to make it a bit more plausible.

Let's assume that we could declare a discriminated union in C# like this:

```
public struct enum Shape  // use the keywords struct enum to identify a structured enum
{
    Rectangle(float width, float length),  // inherits the visibility of the parent Shape (
    Circle(float radius),
    Triangle(float width, float height)
}
```

And the pattern matching could be exposed like this:

```
public static float CalculateArea(Shape shape)
{
    switch shape
    {
        Rectangle(w,l) =>  return w * l;  // deconstruct the union to "local variables"
        Circle b =>  return 2.0f * Math.PI * b.radius;  // no deconstruct, simply use it v
        Triangle(w,h) =>  return w * h / 2.0f;
    }
}
```

They would also support generics as well so it should be possible to declare such a type:

```
public struct enum Option<T>
{
  None,
  Some(T value)
}
```

Projection to .NET

We would like to have a compact, lightweight valuetype to implement discriminated unions and this is where my previous post "**Struct inheritance in C#**" comes into play.

The main idea of the implementation is to collapse/compact all fields of the union types (e.g `Rectangle(width,length)`) into the base union type (e.g `Shape`). This is similar to the way **Rust enum shines**.

So internally, the `Shape` discriminated union would be represented like this:

```
[Union]
public struct Shape
{
    internal const int $RectangleType = 1;
    internal const int $CircleType = 2;
    internal const int $TriangleType = 3;
    private Shape() {} // Note this is currently impossible for a struct
    internal int $type;
    internal float $a,$b,$c;
}
```

and the `Rectangle` type could be mapped to this kind of code:

```
[Union]
public struct Rectangle : Shape
{
    public Rectangle(float width, float length)
    {
        $type = $RectangleType;
        $a = width;
        $b = length;
    }

    public width => $a;
    public length => $b;
}
```

It means that whatever the subtype is, you will always get the same size for a class of structured enum. So you could for example allocate an array of them and they would layout as a plain array of valuetypes. The subtypes (e.g `Rectangle`) would not contain any additional fields: This is where **virtual structs are really shining**.

I have omitted all the other verbose code parts (`Equals`, `HashCode`, `ToString`...etc) that must be implemented for a full implem.

Note that in order to make the size of the base class the most compact, we would need to **reorder and collapse fields into compatible/blittable types**. With a struct explicit layout, it would be straightforward. Typically, if two union types are using a managed object as part of their respective data, we should only store a single `object` reference to the base, so typically:

```
public class MyClass1 {}
public class MyClass2 {}

public struct enum BaseType
{
   Type1(MyClass1 class1),
   Type2(MyClass2 class1),
}
```

would collapse internally to a single object field in the base `BaseType` storage implementation:

```
[Union]
public struct BaseType
{
    internal const int $Type1Type = 1;
    internal const int $Type2Type = 2;
    internal int $type;
    internal object $a; // Class1 and Class2 collapsed into this
}
```

This layout would be compatible with how the GC works to iterate on the reference types stored in the struct, so it would work just as-is.

This field squashing process should be applied also to plain blittable types (primitives, struct...etc.). As you can imagine, there would be some corner cases where an optimal squash would not be entirely possible (like using an existing Struct containing its own field's layout used in a union type parameter)

The pattern matching would then be implemented using a plain switch case:

```
public static float CalculateArea(Shape shape)
{
    switch(shape.$type)
    {
        case Shape.$RectangleType:
        {
            ref var a = ref (Rectangle)shape;
            return a.width * a.length;
        }
        case Shape.$CircleType:
        {
            ref var b = ref (Circle)shape;
            return 2.0f * Math.PI * b.radius;
        }
        case Shape.$TriangleType:
        {
            ref var c = ref (Triangle)shape;
            return c.width * c.height;
        }
        default: // note: more on this later
            return 0.0f;
    }
}
```

In the example above, I have chosen to use an (unsupported?) syntax for `ref` locals that would allow to cast the type, without having to copy it again on the stack. This optimization would be critical in order to avoid any kind of stack copies when deconstructing a `Shape` base type into an inherited union type.

As you can see, using valuetypes is giving a lot more opportunities for efficiency. With the Struct Inheritance, it would be fairly easy to prototype directly such a scenario.

The dirty details

While it sounds very straightforward to implement, there are many pitfalls and languages issues to consider.

Most notably we can try to address/mitigate some of them:

Default value

In .NET, a struct is automatically initialized to zero. Thus, if you have a struct field, it would be automatically initialized ot zero and you would not have to worry about assigning a value to it.

With our structured enum valuetype, we would have a problem: The type `Shape` above would be in a `nil` state if it is not assigned by a sub-Shape (e.g `Rectangle`)

We could circumvent a bit some misuses by forbidding constructions like these:

```
var shape = new Shape(); // Would not compile
Rectangle rect; // Would not compile alone without a constructor

Shape rect = new Rectangle(1,2); // Ok
```

This is fine for local variables, but more problematic if such a type is embedded into another struct.

Thus we would have a state in which a structured enum could be in a `null` state (or `nil` or `none`). Using `null` is not ideal, because the semantic is usually used for reference types, but let's assume that we could use it:

```
Shape shape;

// here shape would be equivalent to `null`
if (shape == null)
{
  shape = new Rectangle(1,2);
}
```

In the pattern matching case, we would have to add a special case for this:

```
public static float CalculateArea(Shape shape)
{
    switch shape
    {
        Rectangle(w,l) =>  return w * l;  // deconstruct the union to "local variables"
        Circle b =>  return 2.0f * Math.PI * b.radius;  // no deconstruct, simply use it v
        Triangle(w,h) =>  return w * h / 2.0f;
        null => return 0.0f; // case for a non initialized entry
    }
}
```

We could also alias null to a special enum type, typically in the case of `Option<T>`, obviously constrained to one that doesn't have any parameters:

```
public struct enum Option<T>
{
  null as None,
  Some(T value)
}
```

behind the scene, it would resolve `None` for `$type` field of the base `Option<T>` to 0 instead of 1. This would allow something like:

```
Option<int> x;

if (x is None)  // equivalent to x == null
{
...
}
```

Array allocation would suffer the same issue and could be resolved the same way:

```
var shapes = new Shape[10];

if (shapes[0] == null)  // this would be case, as shapes[0] hasn't been initialized
{
...
}
```

ref and out aliasing

Similar to the struct inheritance pitfalls, cares must be also taken with the usage of `ref` and structured enums.

Typically, a case like:

```
public float Transform(ref Shape shape)
{
   shape = new Rectangle() // Not ok
   ...
}

Circle circle = new Circle(1.0f)

Transform(ref circle);
```

The `ref` would allow to pass a reference, but this should not allow to assign it completely. This would be similar to an implicit `ref readonly` (as discussed **here**)

On the other hand, and `out` could allow such downcast aliasing:

```
public void Create(int param, out Shape shape)
{
   if (param  > xxx)
   {
      shape = new Rectangle() // Ok
   }
   ...
}

var shapes = new Shape[10];
Create(11, out shapes[0]); // ok
```

but would not allow upcast aliasing:

```
public void Create(out Shape shape)
{
   shape = new Rectangle() // Ok
   ...
}

Circle circle;
Create(out circle); // Not Ok
```

Pinning, Marshalling

How would react a perfectly blittable structured enum to a `fixed` pinning? How would marshal such a type to an interop layer?

Unsure if we should allow such a thing... The projection details is quite implementation dependent, so I don't expect structured enums to interop/marshal to native code.

Next?

Currently, when I have to develop such a behavior in C#, I need to roll my own "freaking" struct, but of course, it cannot be as nice and convenient as the syntax presented here.

As you can see, there are some pitfalls that should be addressed, though I haven't come to an issue that would make this implementation completely impossible/unsafe, but maybe there is one?

Unlike the issue "**Request: make C# 7's discriminated unions compatible with F#**", I really don't want to have this compatibility with F# as it would completely obliterate the performance and efficiency that C# deserve.

Bottom line is: I would love such a type companion in my favorite language toolbox!

So what do you think about the concept?

Copyright © 2009 - 2021, Alexandre Mutel - Blog content licensed under the Creative Commons CC BY 2.5 | Site powered by lunet