

F# metaprogramming and classes

Tomáš Petříček (tomas@tomasp.net)

Introduction

F# quotations allows you to easily write programs that manipulate with data representation of program source code. If you're not familiar with quotations I recommend reading my previous article [\[1\]](#) that contains short introduction to this topic first. Quotations can be used for example for translating subset of the F# language to another code representation or another language.

To get the quotation data of the expression you can either use `<@ .. @>` operator or `resolveTopDef` function. In the first case the code written between the "`<@`" and "`@>`" is converted to data during the compilation. The `resolveTopDef` function allows you to get quotation data of top-level definition (function) from compiled library at runtime (you have to use `--quotation-data` command line switch while compiling the library). I mentioned that quotations can be used to represent only subset of the F# language. Currently, one of the quotation limitations is that it's not possible to enclose the whole class in the quotation operators. It is also not possible to get the representation of the whole class at runtime nor the representation of class members (for example methods).

Class quotations

In one my project I wanted to be able to translate the class written in F# to another language, so I wanted to make this possible. This option will be probably implemented in the future versions of F#, so it will be possible to do this using more elegant method, but if you want to experiment with quotations and use classes, you can use my solution to implement the prototype.

In the prototype I implemented, you have to write the class and one module that contains the actual code for the methods. This makes it possible to get the structure of the class using standard .NET reflection classes and to extract quotation data for the class members using F# `resolveTopDef` function. Of course, you can't create instance of module, but for metaprogramming we only need to be able to get the quoted code and unless you want to use the class from code, you can leave the implementation of the class methods empty.

First I had to design the data structures for representing the class. If I were adding this feature to the F# metaprogramming, I would probably extended the `expr` data type to make this possible in consistent way. However I didn't want to modify the F# source code, so I designed the following types that are similar to the `expr` type (I reversed the order of type declarations, so it is easier to understand):

```
/// Structure that contains information about class -  
/// consists of name, name of base class and members  
type classInfo = {  
    name:string;  
    baseName:string option;  
    members:classMember list; };;
```

```

/// Member of the class
/// For working with this type use cmfCtor, cmfMethod, cmfField, cmfProp..
type classMember;;

/// Type for working with class members
/// Query function tests whether member is of specified type
/// Make function creates member
type 'a classMemberFamily with
  member Query : classMember -> 'a option
  member Make : 'a -> classMember
end;;

/// Constructor declaration
/// - parameter should be lambda expression
val cmfCtor : (expr) classMemberFamily;;

/// Method delcaration
/// - method name, expression (should be lambda) and return type
/// (types of parameters are stored in lambda expression)
val cmfMethod : (string*expr*Type option) classMemberFamily;;

/// Field declaration
/// - field name, type and init expression
val cmfField : (string*Type*expr option) classMemberFamily;;

/// Property declaration
/// - property name, getter and setter (both should be lambda expr) and type
val cmfProp : (string*expr*expr*Type) classMemberFamily;;

```

This representation is still very limited - for example it isn't possible to represent polymorphic methods. In the previous code, the `classMember` can be used to represent any member (similarly to the F# `expr` that can represent any expression). The `cmf[Something]` values are equivalent to the `ef[Something]` from F# quotation library and allows you to work with `classMember` type.

The following example shows how to write simple class with module that can be used for extracting the quotation data:

```

#light

// Module with implementation of methods
module Person_Meta = begin
  // Simulates field of the class
  let name = ref ""

  // Represents constructor
  let ctor (n) =
    name := n

  // Represents method
  let Say (pre:string) =
    let s = pre^", my name is "^(!name)^"." in
    print_string s

  // Represents property
  let get_Name () =
    (!name)
  let set_Name (value) =

```

```

    name:=value
end

// The real class
// Members are just a placeholders and the quotations
// are extracted from the previous module
type Person = class
    val mutable name : string;
    new((n:string)) = { name = ""; }
    member this.Say(pre:string) = ()
    member this.Name with get() = "" and set((v:string)) = ()
end

```

In the last code sample, I'll show how the functions I wrote can be used for working with the previous piece of code. To get the representation that I mentioned earlier of the class `Person`, you can use function `getClassFromType`. This function extracts structure of the class (from the class itself) and quoted code from the module with the `_Meta` suffix. The following example is quite simple. It prints basic class information (name and base name) and then iterates through all the members and prints all available information for every member. For printing the `expr` type (which represents the quoted code) I used `printf` function with the `output_any` formatter.

```

#light
do
    // Get the class info for class 'Person'
    let clsInfo = getClassFromType ((typeof() : Person typ).result)

    // Print name and optional base name
    Console.WriteLine("Class '{0}':", clsInfo.name)
    if (clsInfo.baseName <> None) then
        Console.WriteLine(" base: {0}", clsInfo.baseName)

    // Iterate through class members
    clsInfo.members |> List.iter ( fun m ->
        match cmfMethod.Query m with
        | Some (name, expr, ret) ->
            Method "printf ( %s : %a ) = " name output_any ret
        | _ ->
        match cmfField.Query m with
        | Some (name, typ, init) ->
            printf "Field ( %s : %a ) " name output_any typ
        | _ ->
        match cmfProp.Query m with
        | Some (name, getter, setter, typ) ->
            printf "Property ( %s : %a )\n" name output_any typ
            printf "get = %a\n" output_any getter
            printf "set = %a\n\n" output_any setter
        | _ ->
        match cmfCtor.Query m with
        | Some (expr) ->
            printf "Ctor = %a\n\n" output_any expr
        | _ ->
            failwith "Error!"
    )

```

Conclusion

The aim of this article isn't to present fully working solution, but to suggest a few enhancements to the F# metaprogramming that I think would be useful. You can use the source

code attached to this article to find (and experiment with) some interesting use cases of metaprogramming that require working with classes, for example translating classes written in F# to another language. The biggest limitation of the solution I presented is, that you have to write every implementation twice - it occurs in the body of the class and in the module used for metaprogramming too (however the code looks very similar). If you want to translate F# code, you don't need to implement methods/properties of the real class, but if the program creates instances of the class at runtime (and uses the quoted code to perform some analysis etc.), you'll need to write both implementations.

Downloads

- [Download the source code and examples](#)

Links and references

- [1] [F# - Simple quotations transformation](#) [△] - My F# Notes