Datatype Generic Programming in F#

Ernesto Rodriguez and Wouter Swierstra

Workshop on Generic Programming, 2015
This talk

There are numerous libraries for generic programming in Haskell.

• How can we transfer this technology to other languages?
• What limitations do we encounter?
• Can we retain type safety?
About F#

• F# is a functional language, similar to ML
• Runs on the .NET platform
• Pragmatic combination language features, drawing from both object oriented and functional languages.
Functional *and* object oriented

- inheritance and classes;
- reflection mechanism from .NET;
- parametric polymorphism;
- ad-hoc polymorphism;
- algebraic data types and pattern matching;
- first-class functions...
Can we use these features to implement a library for datatype generic programming in F#?
Datatype generic programming in Haskell

1. A representation type or universe

2. A methodology for defining functions by induction over this universe

3. Automatically generated conversion functions converting user-defined datatypes to their generic representation.

We'll start by reviewing the Regular library.
Regular: universe

The Regular universe defines a collection of types used to represent simple algebraic data types:

```
data U t = U
data K a t = K a
data I t = I t
data (a :+: b) t = Inl a | Inr b
data (a :+: b) t = a :+: b
```
Regular: defining generic functions

Generic functions are declared by introducing a new class:

```haskell
class GSum f where
  gsum : f -> Int
```

And instances for the types we saw previously:

```haskell
instance GSum (U t) where
  gsum _ = 0

instance (GSum a, GSum b) => GSum (a :*: b) where
  gsum (x :*: y) = gsum x + gsum y
...
```
Regular: converting to the generic representation

class Functor (PF a) => Regular a where
    type PF :: * -> *
    from : a -> PF a a
    to : PF a a -> a

sum :: Regular a => a -> Int
sum x = gsum (from x)

Instances the Regular class for user-defined types are typically generated using Template Haskell.
Porting these ideas to F#

To write a library for datatype generic programming in F# we'll need to define the following three ingredients:

1. A representation type or universe

2. A methodology for defining functions by induction over this universe

3. Automatically generated conversion functions converting user-defined datatypes to their generic representation.
Representation types in F# – I

We will use an F# class to define our representation types:

```fsharp
[<AbstractClass>]
type Meta () = class end
```

We can now define subclasses for each of the type constructors we wish to support in our universe.
Representation types in F# - II

All subclasses of the Meta class take an additional phantom type argument, ty, recording the type being represented:

type U<`ty>() =
  class
    inherit Meta()
  end

type K<`ty,`x>(elem : `x) =
  class
    inherit Meta()
    member self.Elem
      with get() = elem
  end
Note that types stored in `Sum` or `Prod` must be subtypes of `Meta`. 
Why do you need to use classes?
Defining generic functions

We would like to use F#'s ad-hoc overloading to define generic functions, just as we used Haskell classes previously:

```fsharp
type Prod<'t,'a,'b when 'a : (member GSum : int) and 'b : (member GSum : int) > with
    member self.GSum = self.E1.GSum + self.E2.GSum
```

Unfortunately, this style of generic function definition does not work well...
Restriction's on ad-hoc overloading

- No overlapping instances
- F# needs to know statically how all overloading is resolved
- Member functions defined post-hoc with an extension are not checked when solving member constraints

F#'s treatment of overloading is very different Haskell type classes
Our approach

Instead of using overloading, we provide an (abstract) class FoldMeta that:

• collects the required definitions for the constructors of our universe

• provides a function that servers as a workaround to handle some of these limitations.
FoldMeta

AbstractClass
type FoldMeta<\`t,\`inp,\`out>() =

abstract FoldMeta : Meta * `inp -> `out
abstract FoldMeta<`ty> : Sum<`ty,Meta,Meta> * `inp -> `out
abstract FoldMeta<`ty> : Prod<`ty,Meta,Meta> * `inp -> `out
abstract FoldMeta<`ty,`a> : K<`ty,`a> * `inp -> `out
abstract FoldMeta : Id<`t> * `inp -> `out
abstract FoldMeta<`ty> : U<`ty> * `inp -> `out
Defining GMap

type GMap<`t,`x>() =
    class
        inherit FoldMeta<
            `t,
            `x -> `x,
            Meta>()
        ...
    end

Defining GMap - products

```haskell
override x.FoldMeta`ty>
  (v : Prod`ty,Meta,Meta>
  ,f : `x -> `x) =
  Prod<Meta,Meta>(
    x.FoldMeta(v.E1,f),
    x.FoldMeta(v.E2,f))
  => Meta
```

Note: we need to cast the result back to a value of type Meta

Also note: recursive calls happen on values of type Meta
Defining GMap – constants

We provide two definitions for the K type:

\[
\text{member } x.\text{FoldMeta}<\text{ty}>(v : K<\text{ty},x>, f : x->x) = \\
K(f v.\text{Elem}) :> \text{Meta} \\
\]

\[
\text{override } x.\text{FoldMeta}<\text{ty},a>(k : K<\text{ty},a>,f : x -> x) = \\
k :> \text{Meta} \\
\]

The override is required and leaves the value unchanged;

The member function works specifically for values of type \(x\) and applies the argument function.
Resolving overloading

Recall how recursive calls happen on values of type Meta – but we have only provided definitions for specific types, such as sums, products, and constants.

Similarly, we have provided *more than one* definition for constants.

How is this overloading resolved?
FoldMeta again

The FoldMeta class has one additional function:

FoldMeta : Meta * `inp -> `out

This method should not be overridden by the user.

Instead, it handles the selection of the right overloaded method.
Implementation

• The implementation of this FoldMeta function is fairly messy.
• It uses .NET reflection to check the type of its Meta argument
• And calls the most method with the most specific that will still accept this argument.
• The good news: users never have to see the reflection code.
• The bad news: there is a run-time penalty in every step of the execution of a generic function
Porting these ideas to F#

To write a library for datatype generic programming in F# we'll need
to define the following three ingredients:

1. A representation type or universe

2. A methodology for defining functions by induction over this universe

3. Automatically generated conversion functions converting user-defined datatypes to their generic representation.
Generating conversions

We can generate conversions using the .NET reflection mechanism.

Every .NET value has a member function:

Get Type : unit -> Type

F# extends the Type class with specific information for algebraic data types.

This allows us to lookup the constructors of a data type, their types, etc.
Generating conversions

In contrast to Haskell, this meta-programming is done at run time.

It is untyped and requires a lot of boilerplate code.

It requires a lot of .NET expertise.

It's not cross platform.
Generating conversions

Nonetheless, we can provide an automatically generated conversion function to the Meta representation type:

```fsharp
type Generic<`t>() =
    member x.To : `t -> Meta
    member x.From : Meta -> `t
```
Top-level function

Now we can use the GMap $\Rightarrow$ FoldMeta class to define the following |gmap| function:

```
member x.gmap(x : t,f : `x -> `x) =
    let gen = Generic<`x>()
    x.FoldMeta(gen.To x,f)
    |> gen.From
```
Taking stock

1. A representation type or universe

2. A methodology for defining generic functions

3. Automatically generated conversion functions converting user-defined datatypes to their generic representation.
Universe definition

We can mimic the Regular universe using classes and subtyping.

This allows us to represent the same collection of types in F# as you can in Haskell.

Allows us to exploit subtyping – bundling the type constructors, rather than define them individually as in Haskell.
Defining generic functions

• The generic functions themselves are 'unityped' – they all manipulate Meta values

• This may cause run-time failures when converting back to user-defined data types.

• We can only handle folds over generic types.

• But we can provide variations of FoldMeta to work on more than one argument, generate Meta values, etc.
Generating conversions

We can use .NET to generate conversion functions.

It's a bit messy, but it works.

These conversion functions are generated at run-time – memoization might really help improve performance.
Advantages over Regular

A generic function is determined by our FoldMeta class.

We can use OO overriding and inheritance to create variations of existing generic functions:

type ShallowGMap<'t,'a>(f : 'a -> 'a) =
  inherit GMap<'t,'a>(f)
  override self.GMap(id : Id<'t>) = id
Conclusions

• We can port many ideas from the datatype generic programming in Haskell to F#

• But we sometimes end up fighting the type system, rather than exploiting it.

• The library provides a more lightweight alternative to existing approaches to generic programming that rely heavily on reflection.
Future work

• We could use reflection (once again) to perform static analysis on compiled assemblies to check the type safety of generic definitions.

• Memoization of conversion functions

• Explore alternative approaches to datatype generic programming that might be easier to adopt in F#.
Uniplate

Using this library, we can support other styles of generic programming such as Uniplate.

\texttt{uniplate} : Uniplate \( a \Rightarrow a \rightarrow ([a], [a] \rightarrow a) \)

Several traversals, transformations and generic functions can be built on top of this.
Uniplate example

```haskell
type Arith =
    | Op of string*Arith*Arith
    | Neg of Arith
    | Val of int

let (c,f) = uniplate (Op ("add",Neg (Val 5),Val 8))

-- prints [Neg (Val 5);Val 8]
printf "%A" c

-- prints Op ("add",Val 1,Val 2)
printf "%A" (f [Val 1;Val 2])
```
Uniplate in F

We can define uniplate using two generic helper functions:

• collecting subtrees
• reconstructing trees
type Collect<'t>() =
    inherit FoldMeta<'t,'t list>()

override self.FoldMeta<'ty,`a>(_ : K<'ty,`a>) = []

override self.FoldMeta<'ty>(_ : U<'ty>) = []

override self.FoldMeta(i : Id<'t>) = [i.Elem]
Collecting subtrees - II

```plaintext
override self.FoldMeta<'ty>(
    c : Sum<'ty,Meta,Meta>) =
    match c.Elem with
    | Choice1Of2 m -> self.Collect m
    | Choice20f2 m -> self.Collect m

override self.FoldMeta<'ty>(
    c : Prod<'ty,Meta,Meta>) =
    List.concat<'t> [
        self.Collect c.E1
        ; self.Collect c.E2]
```

Constructing subtrees - I

type Instantiate<`t>(values` : `t list) =
  inherit FoldMeta<`t,Meta>()
  let mutable values = values`

  let pop () = match values with
   | x::xs -> values <- xs;Some x
   | [] -> None

  override self.FoldMeta(i : Id<`t>) =
    match pop () with
    | Some x -> Id<`t>(x)
    | None -> failwith "Not enough args"
    :> Meta
Constructing subtrees - II

```java
override self.FoldMeta<`ty>(
    p: Prod<`ty,Meta,Meta>) =
:> Meta

override self.FoldMeta<`ty>(
    s : Sum<`ty,Meta,Meta>) =
match s with
| Choice1Of2 m -> Sum<`ty,Meta,Meta>(
    self.FoldMeta m |> Choice1Of2)
| Choice2Of2 m -> Sum<`ty,Meta,Meta> (
    self.FoldMeta m |> Choice2Of2)
:> Meta
```
If you squint enough, it looks just like Haskell
Questions?