Generic Programming in F#

Ernesto Rodriguez

Computer Science
Utrecht University
Utrecht
The Netherlands

Type: Master's Thesis Proposal
Date: November 28th, 2015
Supervisor: Dr. Wouter Swierstra

---

1 Representing Algebraic Data Types

Functional languages use algebraic data types as a specification for data structures. Normally, the programmer uses pattern matching to deconstruct an algebraic data type in order to write algorithms. In order to define generic algorithms that work on any ADT[1], approaches such as Generic Deriving [?], Regular [?] and RepLib [?] provide a collection of types to which ADTs can be converted back and forth. Algorithms are then specified in terms of theese "universal" types. The approach used by Regular is explained in the follwoing section.

1.1 Representations with Regular

A selection of types are used to encode data into representations. Theese types are designed to provide a type-safe mechanism that performs the translation. In other words, the compiler an check wether a particular representation is a valid representation for a particular type. Those types encode the structure of a type as well as some meta-information about such type. Regular uses the following types:

```
data Unit r = Unit
data K a r = K a
data (f ⊕ g) r = Inl (f r) | Inr (g r)
data (f ⊗ g) r = f r ⊗ g r
data Id r = Id r
```

These constructs correspond to the syntax of an ADT as follows:

- `Unit` corresponds to constructors that take no arguments (ie. `Nothing` or `Nil`).
- `K` corresponds to constructors that take one argument and singleton values (ie. `Just`).
- $\oplus$ to sum of two constructors. Denotes that a type can be defined by either of the constructors represented by each of $\oplus$'s arguments.
- $\otimes$ to the product of constructors. Denotes that a type is constructed out of multiple components (ie. `Cons` requires a value and list).
- `Id` represents recursion within the type (ie. a type defined in terms of itself like a list).

Concretely speaking, lets take a look at the `List` type and its representation:

[1] Some types (like Generalized Algebraic Data Types [?] cannot be represented

```
data List a = Cons a (List a) | Nil

instance Regular (List a) where
  type PF (List a) = K a ⊗ Id ⊕ Unit
```

The algorithms that perform the actual encoding/decoding are the following:

```
instance Regular (List a) where

  from (Cons x xs) = Inl (K x ⊗ Id xs)
  from Nil = Inr Unit

  to (Inl (K x ⊗ Id xs)) = Cons x xs
  to (Inr Unit) = Nil

from (Cons 1 Nil) ≡ Inl(Prod((K 1) (I Nil))) -- True
```

The mapping is very straightforward since each of the type constructors is represented as nestings of the `Inl` and `Inr` constructors. This process can be easily automated. In fact Generic Deriving [**?**] is bundled as an extension for the Glassgow Haskell Compiler (GHC) which automatically derives such definitions.

Generic functions can now be specified over those representations. Take for example the generic map function `gmap` which takes a function of type `Int` → `Int` and applies the function to every integer in the structure:

```
gmap :: (GMap (PF a), Regular a) ⇒ (Int → Int) → a → a
gmap (+1) (Cons 1 (Cons 2 Nil)) ≡ (Cons 2 (Cons 3 Nil))
gmap (+1) (Just 1) ≡ Just 2
gmap (+1) Nothing ≡ Nothing
```

To define such a function, one needs to specify how it should operate on each of the types that make up a representation. This requires function overloading which is done in Haskell via Type-Classes. First a type-class with the specification of the operation is defined. In the running example it will be called GMap. It's method `gmap'` takes as an argument a representation and returns a matching[2] representation. The class is the following:

```
class GMap f where
  gmap' :: (GMap (PF r), Regular r) ⇒ (Int → Int) → f r → f r
```

Now each of the types that make up a representation are made an instance of the class. Since the `GMap` constraint is present for the types that have other representations nested inside them. This allows to recursively call the `gmap'` function, no matter what the nested types are. The compiler cannot statically determine which of the `gmap'` overloads will be invoked until it knows what the argument is[3]. In other words, the functions that get called depend on the resulting type after the compiler instantiates all the polymorphic types in a call site.

```
instance GMap U where
  gmap' _ _ = U

instance GMap (K Int) where
  gmap' f (K i) = K (f i)

instance GMap (K a) where
  gmap' _ (K a) = K a

instance (GMap g, GMap h) ⇒ GMap (g ⊕ h) where
  gmap' f (Inl a) = Inl (gmap' f a)
  gmap' f (Inr a) = Inr (gmap' f a)
```

[2] According to the type of the representation
[3] Contrary to F# where all method calls must be resolved statically

```
instance (GMap g, GMap h) ⇒ GMap (g ⊗ h) where
  gmap' f (g ⊗ h) = gmap' f g ⊗ gmap' f h

instance GMap Id where
  gmap' f (Id r) = (Id ∘ to ∘ gmap' f ∘ from) r
```

With the specification in place, the compiler can use type level computations to resolve what are the exact overloads of `gmap'` that must be invoked whenever the `gmap'` is applied. Note that this code contains "Overlapping Instances" which are not Haskell 2010 standard. GHC in particular deals with such instances by choosing the more specific one[4]. To complete the definition, the encoding/decoding functions have to be attached to build the final master function:

```
gmap f = to ∘ gmap' f ∘ from
```

The experienced Haskeller should take a moment to appreciate all the "unique" language features that were required for this apparently simple task. They will be addressed in more detail in the following sections.

1.2 Abandoning Type-Safety

The approach from Regular will be used as a starting point to introduce generic programming into F#. However the approach cannot be directly translated. Following paragraphs outline the problems.

Given the absence of type-classes, overloading in F# must be done by overloading methods of a class. One can start with a definition of `GMap` that looks as follows:

```
type GMap() =
  class

    //Corresponds to K
    member this.gmap'<'t> : (K<'t> * (int → int)) → K<'t>

    //Corresponds to U
    member this.gmap' : (U * (int → int)) → U

    //Corresponds to I
    member this.gmap'<'t> : (Id<'t> * (int → int)) → Id<'t>

    //Corresponds to ⊗
    member this.gmap'<'a,'b> : (Prod<'a,'b> * (int → int)) → Prod<'a,'b>

    //Corresponds to ⊕
    member this.gmap'<'a,'b> : (Sum<'a,'b> * (int → int)) → Sum<'a,'b>

  end
```

This definition provides an overload of the `gmap` function for each of the constituents that define a type representation. It is important to note that each of the overloads must *somehow* be able to recursively[5] invoke the appropiate overload depending on what type arguments are used to call the `gmap` function.

Speaking concretely, suppose one has the representations τ_1 and τ_2 defined as `Prod<K<int>,Prod<K<string>,U>>` and `Prod<K<int>,U>` respectively. Suppose `gmap` is invoked with τ_1 as argument. Here the type parameters `'a` and `'b` get instantiated to `K<int>` and `Prod<K<string>,U>` respectively. When the time comes to invoke `gmap` with the value corresponding to `'b`, the correct overload (the one for the `K` case) must be selected. Now consider invoking `gmap` with τ_2 as argument. When `gmap` gets invoked with the value corresponding to the type parameter `'b`, a *different* overload (the case for `U`) is selected. This subtle difference makes use of one of Haskell's type system luxuries which selects a different overload depending on how the types variables get

[4] Or failing if two instances are "equally" specific

[5] It is not strictly speaking recursion since a different method with the same name is being invoked

instantiated. However, this is not possible in F#. The overload to be invoked must be known soley from the definition of the method.

This imposes the restriction that all the type variables must be instantiated to some *common* type that rules them all. For that purpose, the class `Meta` is defined. This allows all *recursive* calls to refer to a unique overload. In other words, the class hierarchy now looks like:

```
K<'t> :> Meta

Prod<'a,'b> :> Meta

Sum<'a,'b> :> Meta

U :> Meta

Id<'t> :> Meta
```

where `:>` corresponds to the sub-class relation.

With this adjustment in plae, the definition of `gmap` could be the following:

```
type GMap() =
  class

    //Corresponds to K
    member this.gmap'<'t> : (K<'t> * (int → int)) → K<'t>

    //Corresponds to U
    member this.gmap' : (U * (int → int)) → U

    //Corresponds to I
    member this.gmap'<'t> : (Id<'t> * (int → int)) → Id<'t>

    //Corresponds to ⊗
    member this.gmap' : (Prod<Meta,Meta> * (int → int)) → Prod<Meta,Meta>

    //Corresponds to ⊕
    member this.gmap'(Sum<Meta,Meta> * (int → int)) → Sum<Meta,Meta>

  end
```

This approach solves ambiguity of deciding which is the overload that should be invoked. Nevertheless, it is now required to have an overload to handle the case for `Meta` since that is the overload that will be selected when performing recursive calls. It looks as follows:

```
type GMap() =
  class
    //Corresponds to Meta
    member this.gmap' : (Meta * (int → int)) → Meta
    ...
  end
```

What should be the implementation of this method. Recall that the reason this method is required is because F# dosen't have a mechanism to select the right overload according to the instantiation of the type variables. This means that the **sole** purpose of this method is selecting the overload that must be called. This mechanism will be described in-depth in the next sections.

1.3 Type-Representations in F#

Before exploring solutions to the problems mentioned above, this section explains in depth how the ADTs will be precisely represented in F#. A class hierarchy is used to achieve it. On the top of the hierarchy is a class called Meta with no type arguments. This class is the mother of all representations. There are 6 sub-classes of this class:

- U with no type arguments which is analogous to the Unit in regular.
- K<'t> with the type argument 't which is analogous to K in regular. The type argument denotes the type of its contents.
- Prod<'a,'b> which is analogous to $\otimes$ in regular. It's type arguments are the types of the product and they are required to be a sub-class of Meta.
- SumConstr<'a,'b> wich is analogous to $\oplus$ in regular. From this class, the subclasses L<'a,'b> and R<'a,'b> are defined (corresponding to the left and right branches) which are the ones actually used to create the representation.
- I<'t> wich is analogous to I in regular. This class is used to represent recursion within a type.

The encoding process is very straightforward, below each of the cases is described. The algorithm is described in reference to the input type which is denoted by T:

Values:

- If the value is of type T return Id<T>.
- If the value is an ADT, calculate its representation but using 't as the reference type for recursion.
- Otherwise return K<V> where V is the type of the value

Constructors:

- If constructor takes no argumetns return U
- If constructor takes argumetns. 1) Take the first argument and apply the value case. 2) Apply the constructor case to the remaining arguments. 3) Pack the two results in Prod<R1,R2> where R1 is the representation resulting from the first argument and R2 is the representation resulting from the remaining arguments.

Sum of Constructors Constructors for a type will be ordered in some arbitrary deterministic order. A type is constructed as follows:

- For the first constructor, the type for its representation is calculated (C0) and it is then given the type L<C0,U>.
- For the nth constructor, its representation is calculated (Cn) then it is connected to the representation of the previous constructors (Cp) to construct a type SumConstr<Cn,Cp>.

Given a value. It is now possible provide a representation matching the corresponding type. In the previous step, for each constructor a type SumConstr<C1,C2> has been calculated. It is then assigned the type L<C1,C2> (which is a subtype of (SumConstr<C1,C2>) and for each of the constructor that appear above the ordering, a nesting inside the R constructor is performed. This algorithm is implemented in the class Generic<'t> whos type argument corresponds to the input type which is the type that will be given to the Id constructor.

1.4 A New Hope: Type-Providers

There is a mechanism in F# that allows (very wild) type level programming: Type-Providers [?]. Although initally designed to permit typed access to data sources. They allow a lot of fancy declaration of new types (thanks to the excellence of .Net) at compile time.

Type-providers will be used to improve the type-safety of the library. In order to do so, the user must give it as arguments the name of the generic function and the number of arguments (apart from the type-representation) it takes. This will create a type that includes the following:

1. An implementation for the `Meta` variant of the generic function wich will act as a dispatcher which selects the right generic method.
2. Default cases for all the representations.

For the default implementation to work, the user must specify a single parameter to the constructor of the generated type which is a function that will be invoked as default. The follwoing code examplifies how this would be done:

```
type GMapBase = Generics.Provided.Generic<"gmap",1>

GMapBase.new : ((Meta → obj → obj) → GMapBase)

GMapBase.gmap : (Meta → obj → obj)

GMapBase.gmap : (K<obj> → obj → obj)

GMapBase.gmap : (Prod<Meta,Meta> → obj → obj)

GMapBase.gmap : (Sum<Meta,Meta> → obj → obj)

GMapBase.gmap : (Id<obj> → obj → obj)

GMapBase.gmap : (U → obj → obj)
```

The type created by the provider has a default implementation for all the possible representation elements. The implementation for the case that takes `Meta` as first argument selects at runtime the method that better matches the given constructor. It will be explained in later sections how such ordering of types has been defined. For the rest of the cases, the function provided in the constructor will be invoked. Now it is guaranteed that at runtime a method will always exist for any possible representation of a type. The programmer can override theese methods to change the behaviour of the generic function in a subclass. The approach still has the flaw that all arguments (except of the generic constructors) are of type `obj`. One would like something like:

```
type GMapBase<'t,'a,'r> = Generics.Provided.Generic<"gmap",1>

GMapBase<'t,'a,'r>.new : ((Meta → 'a → 'r) → GMapBase)

GMapBase<'t,'a,'r>.gmap : (Meta → 'a → 'r)

GMapBase<'t,'a,'r>.gmap : (K<obj> → 'a → 'r)

GMapBase<'t,'a,'r>.gmap : (Prod<Meta,Meta> → 'a → 'r)

GMapBase<'t,'a,'r>.gmap : (Sum<Meta,Meta> → 'a → 'r)

GMapBase.gmap : (Id<'t> → 'a → 'r)

GMapBase.gmap : (U → 'a → 'r)
```

This would allow a very precise specification of the generic functions. Unfortunately, the types resulting from the invocation of Type-Providers cannot have generic arguments. The reason is that the arguments that type-providers accept are literals (strings, numbers, booleans, etc.) and even after the initial invocation (when the type-synonym is declared) the compiler still treats the resulting type as a type-provider instead of an ordinary type. Due to this shortcomming, it is only possible to implement the first approach.

However, the author of generic algorithms can easily add a bit of extra type safety to a wide variety of generic algorithms. In particular the ones where the additonal arguments are constant such as the `GMap` function. This is done by providing the type parameters on the sub-class that implements the algorithm and requiring that the arguments are provided in the constructor. Additionally, in order for the `Id` branch to allow recursive calls, the class must be given the type parameters corresponding to the type being handled by the function. The code below shows an implementation of `GMap` using the above strategies:

```
type GMapBase = Generics.Provided.Generic<"GMap",0>

type GMap<'t>(g : int -> int) =
  inherit GMapBase(id)

  let generic = Generic<'t>()

  member x.GMap(c : Meta) =
    base.GMap(c) :?> Meta

  member x.GMap(c : L<Meta,Meta>) =
    L<Meta,Meta>(x.GMap(c.Elem))

  member x.GMap(c : R<Meta,Meta>) =
    R<Meta,Meta>(x.GMap(c.Elem))

  member x.GMap(c : Prod<Meta,Meta>) =
    let v1 = x.GMap(c.E1)
    let v2 = x.GMap(c.E2)
    Prod<Meta,Meta>((v1,v2))

  member x.GMap(c : K<int>) =
    K<int>(f c.Elem)

  member x.GMap(c : Id<'t>) =
    Id<'t>(x.GMap(g.To c.Elem) |> g.From)

let gmap<'t,'v> (f : int -> int) (a : 't) =
    let g = Generic<'t>()
    let e = GMap<'t>(f)
    g.To a |> e.Everywhere |> g.From
```

The overload of `GMap` for the case of `Meta` is meerly a conveniece that invokes the selector method and casts the result from `obj` to `Meta`. If generic type argumetns were allowed for type providers, this could be done by the provider for free.

1.5 Selecting the Overload

This implementation still has some mystery on how the overloads are selected. In particular, the F# type system does not allow automatic casting of a type inside a container. In other words: `T<int> :> T<obj>` is not allowed in F# in spite that `int :> obj` is allowed. This implementation has some sub-typing rules that specify how types are related. First, given the `:>` relation already present in F#, the $\sim$ relation is defined:

$$\tau_1 \sim \tau_2 = \begin{cases} \mathtt{true} & \text{if } \tau_1 :> \tau_2 \\ \tau_1' \sim \tau_2' \land \tau_1'' \sim \tau_2'' & \text{if } \tau_1 = C < \tau_1', \tau_1'' > \land \tau_2 = C < \tau_2', \tau_2'' > \land C \in \{\mathtt{L}, \mathtt{R}, \mathtt{Prod}\} \\ \tau_1' :> \tau_2' & \text{if } \tau_1 = \mathtt{K} < \tau_1' > \land \tau_2 = \mathtt{K} < \tau_2' > \\ \mathtt{false} & \text{otherwise} \end{cases}$$

Given this relation, a partial order $\rhd$ between types is defined. The order is partial because two types are comparable in $\rhd$ if they are related by $\sim$. Following is the definition of $\rhd$:

$$\tau_1 \rhd \tau_2 = \begin{cases} \tau_1'' \rhd \tau_2'' & \text{if } \tau_1 = \mathtt{Prod} < \tau_1', \tau_1'' > \land \tau_2 = \mathtt{Prod} < \tau_2', \tau_2'' > \land \tau_1' \equiv \tau_2' \\ \tau_1' \rhd \tau_2' & \text{if } \tau_1 = \mathtt{Prod} < \tau_1', \tau_1'' > \land \tau_2 = \mathtt{Prod} < \tau_2', \tau_2'' > \\ \tau_1' \rhd \tau_2' & \text{if } \tau_1 = \mathtt{L} < \tau_1', \tau_1'' > \land \tau_2 = \mathtt{L} < \tau_2', \tau_2'' > \\ \tau_1'' \rhd \tau_2'' & \text{if } \tau_1 = \mathtt{R} < \tau_1', \tau_1'' > \land \tau_2 = \mathtt{R} < \tau_2', \tau_2'' > \\ \tau_1' :> \tau_2' & \text{if } \tau_1 = \mathtt{K} < \tau_1' > \land \tau_2 = \mathtt{K} < \tau_2' > \\ \tau_1 :> \tau_2 & \text{otherwise} \end{cases}$$

In order to select an overload, the first step is filtering the suitable methdos by comparing the type given as a first argument using $\sim$ and ordering the resulting options with $\rhd$ to select the most specific one. Note that for the case of `Prod`, the ordering is biased because the first type argument is given priority when selecting the specificity of the method.