Generic Programming in F#

Ernesto Rodriguez

Computer Science
Utrecht University
Utrecht
The Netherlands

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

Executive Summary

Often enough the type system gets in the way of the programer and requires him to write several times the same boilerplate code in order for the program to type-check. One particular effort to reduce this inconvenience is Generic programming. Generic programming allows programers to define functions that operate on the strucutre of the data (the representation) such that the same generic functions can work on different types and additionaly the compiler is able to type-check the correctness of the algorithm. This approach has matured over many years in Haskell and proven to be useful in applications such as Quickcheck. The F# programming language would also benefit from Generic programming since currently the only alternative there exists is reflection. Unfortunately, it is not straightforward to translate the existing Haskell approaches to F# since the language lacks features in it's type-syste, in particular higher-kind generics and implicit parameters (Type-Classes). The objective of this research is to investigate the possible approaches for generic programming in F#, taking as inspiration what exists in Haskell, and developing a type-provider that can generate the boilerplate code necessary for Generic programming.

---

1 Introduction

Often enough one desieres to write functions that are heavily dependent on the structure of data and hardly on the type of the data. Such functions are trival with dynamic typing and impossible with static typing leading to multiple implementations of essentially the same algorithm. To address this shortcomming, Generic Programming has been developed, mostly in the context of Haskell. Generic Programming provides the programmer a mechanism to represent types (usually algebraic data types) which he uses to implement algorithms that operate on the representation of the type and are then translated to the original type or the resulting type.

There exists several approaches in Haskell [**?**, **?**, **?**] each of them with their own advantages and limitations. For example Regular [**?**] is a very lightweight library but dosen't suporrt mutually recursive types. MultiRec [**?**] supports mutually recursive types but generic algorithms have an extra degree of complexity. It has been shown that Generic programming works well in Scala [**?**] although it isn't much of a surprise since Scala supports most type-system features in Haskell; the most notable differences are that Scala supports sub-typing (making type inference undecidible) and Scala uses implicit arguments as a replacement for type-classes.

One would expect that the existing literature would naturally translate to the ML dialect F# but F# lacks certain type-system features, in particular higher-kind generics and implicit parameters, which are heavily used for the existing approaches [**?**, **?**, **?**]. The absence of theese features can be tought to be a language maturity issue which, when addressed, could potentially make this work worthless but there are at least two reasons not to expect theese features to ever be part of the F# language:

1. Even though type inference is not decidible in F#, it is much better than it is in Scala and such features could threaten this luxury
2. F# offers complete interoperability with other .Net languages. Theese features would either require changes in the .Net platform and it's guest languages or type erassure (as done in Scala)

2 Background

The following section contains a breif overview of the concepts that will be dealt with during this research. It is by no means comprehensive and the reader is adviced to consult the refrenced literature for more details.

2.1 Overview of Generic Programming

The term Generic Programming is very broad. In [**?**] the following definition is given: "Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures and other software concepts, with their systematic organization". Many approaches exist that fit this deffinition, in particular, the most popular one are the so called "Generics" as described in [**?**] which are a mechanism that allows programming languages to abstract over the type inside a container type.

The present research focuses on a different kind of Generic Programming, namely Data-Type Generic programming as it is described here [**?**]. The term Generic Programming will always refer to Data-Type Generic Programming in this document. The idea behind Generic Programming is to define algorithms that operate on the structure of Algebraic Data Types (ADTs) and some of it's variants.

Even though ADTs are a simple concept, advanced features of higher order programming languages such as parametric polymorphism and higher-kinds impose a lot of difficulties when designing a Generic Programming library. Hardly any of the existing libraries [**?**] supports all the features of the Haskell 98 standard. Based on years of experience of the top researchers of the field, in [**?**] a list of desireable features of a good generic programming library is given. No name a few:

- Size of the universe (how much data-types are supported): Does the library support ADTs with multiple type arguments (ie. ADTs of kind $* \to (* \to (* \to ...)))$? Can the type arguments of the ADTs be higher-kinded? Can the size of the universe be easily extended?
- Expressiveness: Are generic functions first class functions? Are the representations of ADTs as simple as the can be? Does the library provide additional metadata about the ADTs (ie. constructor names, operator fixity, etc.)
- Usability: Is it easy to learn? Does it provide tools to automate the definition of representations? Does it perform well?

For many years, Generic Programing was a dark magic for hard-core Haskell users but lately it has become a standard tool since at least the two main Hakell compilers, namely The Glassgow Haskell Compiler (GHC) and the Utrecht Haskell Compiler (UHC) provide built in support for

generic programming [?, ?].

The Generic Deriving [?] is simple and easy to learn. It's most notable drawback is that it only supports ADTS with a single type argument. Many generic algorithms are still expressible with this restriction making it a very useful library. The author of the document has been developing a small project to experiment with the library [?] and did find the single type-argument restriction problematic for his purposes.

Before introducing in detail how Generic Programming works, the reader should be aware there exists roughly three approaches to develop Generic Programming libraries in Haskell:

- Type-Classes based approaches like Generic Deriving [?], Multi-Rec [?] and PolyP [?]
- Combinator based libraries like SYB [?] and it's variants
- GADTs like RepLib [?]

In the following section we present a breif overview of how Generic Programming works based on the Type-Class approach.

2.2 A breif introduction to Generic Programming

This section introduces Generic Programming as it is done by the Regular [?] library. This library has been chosen because:

1. Generic Deriving which probably the most common implementation is based on Regular
2. It was designed with simplicity in mind

The running example will be implementing the generic function `gmap` for the type `Int`. This function takes a functional argument and applies it to all the integer values of a type. Concretely it looks like:

```
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
```

The type signature includes three constraints which are:

- `Regular` which is the universe of all types that have a representation in Regular
- `GMap` which is a user-defined Type-Class that implements the operation (more about later).
- `PF` which stands for pattern functor is the abstraction used to recurse over sub-trees of the type (ie. recursive types). The details about the pattern functor are presented in [?] but are not necessary to understand the intuition behind Generic Programming.

The magic behind Generic Programming happens with what is called the representation of a type. A representation is meerly a generic mechanism to represent the structure of an ADT. Regular provides the follwoing constructs to build representations:

```
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
```

Theese 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`)
- `:+:` to sum of two constructors. Denotes that a type can be defined either by the constructors represented by either of its arguments.
- `:*:` 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:

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

instance Regular (List a) where
  type PF (List a) = K a :*: Id :+: Unit
```

From this representation it is straightforward to define the `from` and `to` functions which are part of the `Regular` type-class. Theese functions convert data back and forth to/from representations. In fact, Regular provides Template Haskell splices that do it automatically. For completeness, the definition is shown below:

```
instance Regular (List a) where

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

  to (Inr (K x :*: Id xs)) = Cons x xs
  to (Inl Unit) = Nil
```

The next step is writing generic functions that work with representations. The example here will be restricted to types that contain only integers in the `K` constructor but this restriction can be dropped. A complete running example can be found here [**?**].

Regular uses the type-class approach, so the method proceeds by defining a typeclass that performs the operations that interest us and the different pieces of the representation are made instances of that type-class. For the running example, the type-class is defined below:

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

The function `gmap'` of this class works on any container `f` and additonally we require that the contents of `f` are inside the universe of types representable in by Regular (ie. a PF representation has been defined with the `to` and `from` functions). The only reason this restriction is necessary is because lists are recursive types.

To proceed we need to make the containers denoted by `f` instances of GMap. So as expected, the containers that interest us are the types used define the representations. Following are the instances defind for the `GMap` class:

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

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

```
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)

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
```

What is happening here is the following:

- For the `U` case, nothing happens since the constructor contains no values.
- For the `K` case, the function is applied to the contents and then packed again.
- For the `:+:` case, pattern matching is performed to determine which of the constructor cases is being dealt with and `gmap'` is applied to the contents of the constructor. The result is again packed in the same branch case.
- For the `:*:` case, the `gmap'` function is applied to each of the values of the constructor and then packed with the original structure.
- For the `Id` case: The recursive type contained in `Id` is transformed to its representation, `gmap'` is recursively applied and the result is converted back from the representation to a value.

To tie everything up and define the `gmap` function above, we simply wrap `gmap'` with the `to` and `from` functions:

```
gmap f = to ∘ gmap' f ∘ from
```

This concludes the breif introduction to Generic Programming. It meerly scratches the surface of all work that has been done in the field. This implementation has the follwoing limitations:

- Generic types do not abstract over the types of constructor values. This restricts the `gmap` function only to be definable for function arguments of concrete types (ie. `Int`→`Int`) or typeclasses. Using the `Typeable` class, the function can operate with many different types. Such approach is demostrated in [**?**]. Other libraries including Generic Deriving don't have the limitation.
- Due to the restriction above, the `gmap` function cannot change the type of the constructor values
- Producers (such as `read`) for higher-kinded ADTs would be impossible due to the limitation above

3 Objectives and Evaluation

The objectives of this research are the following:

- Develop a motion capture data classifier based on echo state neural networks
- Develop a method to reduce the dimensionality of motion capture data and still archive a successful classification
- Benchmark the method with the data from HDM05 [**?**]

- Compare this method with existing methods, particularly with Motion Templates [**?**] and Self Organizing Maps [**?**]
- Evaluate the possibility of extending the method with automated segmentation

The first aspect to be evaluated is the reduction of dimensionality that will be attempted. It is expected that a classifier can still be built even after reducing the dimensions of the motion capture data. If possible, the classifier will be tested against the performance of a slower classifier which uses the data without reduction.

4 Preliminary Work

An implementation of ESNs based on [**?**] is being developed for this research and the code is available in [**?**]. The code includes functionality to generate ESNs of variable sizes, train the ESNs using the Moore-Penrose psudoinverse and run the ESNs either by teacher forcing or regular means. This implementation has been evaluated for simple series such as sine waves which resulted in good approximations. There is still work to be done since chaotic time series such as the Henon time series do not perform well with this implementation. The final classifier will be based on this implementation which is expected to become more powerful.

5 Timeline

- February 15th, 2013: Complete the development of the code to create, train and run the ESNs and the code used to extract the motion capture data.
- March 15th, 2013: Complete the creation of the expert neural networks.
- April 15th, 2013: Complete testing the classifiers. Measure the performance with aligned and unaligned motion capture data.
- May 15th, 2013: Final report submission.