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

Executive Summary

Type systems are by far the most used static verification tool. Well designed type systems allow the programmer to write code without any extra burden while providing extra safety. Type information can be used to generalize algorithms over large families of structures, this is known as Generic Programming. Generic Programming allows programmers to define functions over the structure of the data (the representation) such that the same generic functions can work on different types; additionally 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 allowing the compiler to automatically derive functions such as binary serialization for different data. 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 its type-system, 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 desires to write functions that are heavily dependent on the structure of data and hardly on the type of the data. Type systems usually have access to the structural information about data-types necessary to write these algorithms. However, such information is usually not easily accessible to the programmer. To address this shortcoming, 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). Then, generic functions are defined by induction on the type structure. On occasions, the result is also a representation of a type which gets translated to the desired resulting type.

There exists several approaches in Haskell [2, 5, 15, 16] each of them with its own advantages and limitations. For example Regular [5] is a very lightweight library but doesn't support mutually recursive types. MultiRec [16] supports mutually recursive types but generic algorithms have an extra degree of complexity. It has been shown that Generic programming works well in Scala [7], although it isn't much of a surprise since Scala supports most type-system features available in Haskell. The most notable differences are that Scala supports sub-typing (making type inference

undecidable) 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#. Unfortunately, F# lacks certain type-system features, in particular higher-kind generics and implicit parameters, which are heavily used by the existing approaches [2, 5, 16]. The absence of these features can be thought to be a language maturity issue which, when addressed, could potentially make this work worthless but there are at least three reasons not to expect these features to ever be part of the F# language:

1. Even though type inference is not decidable 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. These features would either require changes in the .Net platform and its guest languages or type erasure (as done in Scala).
3. Microsoft is not planning to implement higher-kind generics in .Net [14].

The above indicates that if Generic Programming is desired in F# it will have to be implemented with what .Net has to offer. On the other hand, such limitations create an extra desire to have Generic Programming in F# since the inability to abstract over types with a higher-kind leads to boilerplate code.

The present research aims to investigate what is the best approach to allow Generic Programming in the F# language. Thanks to the years of research invested for Generic Programming in Haskell, it is possible to explore multiple alternatives without having to re-invent the wheel. The challenge lies in implementing a library that is as powerful, extensible and easy as the ones in Haskell without all the type-system luxuries of that language.

2 Background

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

2.1 Overview of Generic Programming

The term Generic Programming is very broad. In [1] 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 definition, in particular, the most popular one is the so called "Generics" [1] which is a mechanism that allows languages to abstract over types.

The present research focuses on a different kind of Generic Programming, namely Data-Type Generic programming as it is described here [8]. The term Generic Programming will always refer to Data-Type Generic Programming in the present document. The idea behind Generic Programming is to define algorithms that operate on the structure of Algebraic Data Types (ADTs) and some of its 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 [8] supports all the features of the Haskell 98 standard. Based on years of experience of the top researchers of the field, [8] provides a list of desirable features a good generic programming library should have. No name a few:

- Size of the universe (how many 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 they 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 Programming was a dark magic for hard-core Haskell users but lately it has become a standard tool since at least the two main Haskell compilers, namely the Glasgow Haskell Compiler (GHC) and the Utrecht Haskell Compiler (UHC) provide built in support for generic programming [2, 3].

The Generic Deriving approach [3] is simple and easy to learn. Its 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. Common functions like `map`, `show` or `serialize` are easily expressible in this library. Nevertheless, the author of this document has been developing a small project to experiment with the library [11] and did find the single type-argument restriction problematic for the purpose of writing generic command line argument parsers.

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 [3], MultiRec [16] and PolyP [6]
- Combinator based libraries like SYB [2] and its variants
- GADTs like RepLib [15]

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

2.2 A brief introduction to Generic Programming

This section introduces Generic Programming as it is done by the Regular [5] library. This library has been chosen because:

1. Generic Deriving, which probably is 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 defines the `gmap'` operation (more about it later) on representation of types.

- `PF` (stands for pattern functor) which is the abstraction used by Regular to traverse structures. The details about the pattern functor are presented in [5] 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 merely a generic mechanism to represent the structure of an ADT. Regular provides the following 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
```

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:

```
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. These 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) = Inl (K x ⊗ Id xs)
  from Nil = Inr Unit

  to (Inl (K x ⊗ Id xs)) = Cons x xs
  to (Inr 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 [9].

Regular uses the type-class approach, so the method proceeds by defining a type-class 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 additionally 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 defined 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 $\oplus$ 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 $\otimes$ 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. Then, `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 brief introduction to Generic Programming. It merely scratches the surface of all work that has been done in the field. This implementation has the following 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 type-classes. Using the `Typeable` class, the function can operate with many different types. Such approach is demonstrated in [9]. Other libraries including Generic Deriving don't have the limitation.
- Producers (such as `read`) for higher-kinded ADTs would be impossible due to the limitation above.

2.3 The F# programming languages and Type Providers

The F# [12] programming language is an ML dialect with support for object oriented programming. The language enjoys parametric polymorphism, higher order functions and inheritance/sub-typing. Its most notable limitation compared to Haskell is the lack of support for higher-kinded generics (ie. polymorphism on types with kind $(* \to *)$ and so on).

The language was designed with pragmatism and safety in mind. Type inference, although not decidable, is very complete in the language. The F# code is inter-operable with C# and other .Net languages without the need of type erasure. The philosophy of the language is being a functional first programming language that uses objects and sub-typing whenever convenient.

A particular feature of the F# language is type-providers [13]. Type-providers allow the compiler to obtain types form external sources. An example is obtaining a type that represents the structure of an SQL database by reading it from the database. A type provider can be defined by any end user and used in any program by means of special notation. Internally, a type provider is a function that can inspect the source code of the file in compilation and any other source it wishes and returns one or several F# types. The elegant module system of the language allows modules to be treated as objects so several types can be attached to a module merely by adding members to it.

Even though type providers were not meant to be a meta-programming mechanism. There are no significant restrictions that prevent them from being one. In fact, any type supported by the .Net platform can be generated with a type provider even if it is not definable in F# itself. They are also lazy so they can provide a module that contains infinite number of type definitions and only use the types as they are required in the program.

3 Objectives and Evaluation

The objectives of the present research are the following:

- Research the technologies used by the Generic Programming libraries in Haskell and determine which/how can such technologies can be supported in F#.
- Design and implement one Generic Programming library in F# based on the existing ideas.
- Add support for ADTs of higher kind to the library or outline the limitations that make such library impossible.
- Evaluate whether features (such as inheritance) which are present in F# but absent in Haskell provide extra value in designing Generic Programming libraries.
- Implement a mechanism that can automatically derive the representation of generic types. Possibly using type providers because of their affinity with the language.
- Investigate whether it is possible and valuable to support F# specific augmentations to ADTs with a Generic Programming library.

In the Haskell language, it has been observed that the type-class approach is probably the best method to provide generic programming. It extends nicely [2, 8] and approaches such as SYB had to use it up to some degree to become extensible [2]. Such approach is not directly possible in F# since the language lacks implicit arguments and higher-kind generics. An approach of similar spirit will be considered using interfaces and inheritance to emulate context and allow overloading. Nevertheless, a combinator based approach (and possibly other creative solutions) will also be thoroughly considered because it might happen that they are more suitable for F#. To explore

how these approaches fit in F#, the author will learn about the technologies that have been used to build Generic Programming libraries in Haskell and based on those findings decide which approach will be used to implement a Generic Programming library in F#.

It might not be possible to support ADTs with multiple type arguments in F#. In such a case, the present research intends to outline the type-system limitations of the language that restrict such approaches. On the other hand, the convenience of type providers and programming environments (Visual Studio) of F# might make it possible to support these types in an elegant fashion which would not be practical to implement in Haskell. In the end, F# programmers are not very sensitive to unsafe operations as long as the interface provided is safe. Sub-typing and inheritance also give another dimension of flexibility for F#. For example, in the language one can pattern match over sub-types of a type. This allows the `gmap` function to be defined in a cleaner style [10].

ADTs in F# can be extended with annotations and sub-types. These are available at run-time via reflection and it would be desirable to make them accessible for generic programmers. In the best of cases, the metadata containers of the F# libraries could simply be extended to include such information and expect that it will turn out useful in that way.

Of the criteria presented in [8] extra attention will be giving on having a library that:

1. Supports ADTs with at least one type argument.
2. Allows the universe to be extended in an ad-hoc manner. Without having to re-compile the generic algorithms whenever new types are introduced.
3. Provides first-class generic functions

Achieving #2 is threatened by the fact that F# doesn't support type-classes nor higher-kind generics, both mechanisms are heavily used by the Haskell libraries that are extensible. In the best of cases, using inheritance and interfaces will do the job. Due to this limitations, it is also expected that the libraries in F# will internally require unsafe operations such as casting and type erasure.

4 Preliminary Work

The `gmap` function presented above has been implemented in F# [10] with the use of manually defined `to` and `from` functions. The implementation is still very rudimentary and involves a significant amount of type erasure to operate. Nevertheless, the fact that minimal effort was required is a good indicator that F# allows the implementation of powerful Generic Programming libraries.

5 Work Plan

Date	Objective
February 1st, 2015	Complete the research about technologies used for generic programming. At least MultiRec [16], Uniplate [4], SYB [2] and Generic Deriving [3] will be inspected. Determine which of the technologies employed by the libraries is the best option for a F# library.
March 1st, 2015	Complete an initial prototype for the Generic Programming library. Support at least types without type arguments. The representations will still need to be defined manually.
March 15th, 2015	Complete the type-provider for the prototype. Asses whether type-providers are a sufficient meta-programming mechanism.
April 15th, 2015	Extend the library to support one type argument.
May 1st, 2015	Extend the type provider to support one type argument.
May 15th, 2015	Test what has been implemented so far with common Generic Programming problems. Evaluate the library so far according to the defined assessment criteria.
June 1st, 2015	Determine how to extend the library to support arbitrary number of type arguments.
July 1st, 2015	Implement support for arbitrary type arguments in the library or produce a report indicating why it is not possible.
July 15th, 2015	Extend the type provider to support multiple type arguments. If multiple type arguments are not possible, propose additions that could be made to the F# compiler to make it possible.
August 1st, 2015	Test the final library on common and complex generic programming tasks. Evaluate the library according to the evaluation criteria. Uncover bugs that should be resolved before the final delivery.
August 30th, 2015	Tie the loose ends, fix bugs in the library, write the report and submit the final work.

References

[1] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, March 2007.

[2] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, January 2003.

[3] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[4] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 49–60, New York, NY, USA, 2007. ACM.

[5] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[6] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, pages 168–184, Berlin, Heidelberg, 2004. Springer-Verlag.

[7] Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for generic programmers. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 25–36, New York, NY, USA, 2008. ACM.

[8] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. *SIGPLAN Not.*, 44(2):111–122, September 2008.

[9] Ernesto Rodriguez. Regular - gmap. `https://github.com/netogallo/FSharp-Generics/blob/master/Proposal/src/Main.hs`.

[10] Ernesto Rodriguez. F# - regular gmap example. `https://github.com/netogallo/FSharp-Generics/blob/master/Proposal/Proposal.fs`, 2014.

[11] Ernesto Rodriguez. Kwargs - generic command line arguments parsers. `https://github.com/netogallo/Kwargs`, 2014.

[12] Don Syme. The f# 3.0 language specification. Technical report, Microsoft Research, 2012.

[13] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. F#3.0 - strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[14] Visual Studio Team. Add higher order generics to f# (type classes). `http://visualstudio.uservoice.com/forums/121579-visual-studio/suggestions/2228766-add-higher-order-generics-to-f-type-classes`.

[15] Stephanie Weirich. Replib: A library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 1–12, New York, NY, USA, 2006. ACM.

[16] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. *SIGPLAN Not.*, 44(9):233–244, August 2009.