Datatype Generic Programming in F#

Ernesto Rodriguez
Utrecht University
e.rodriguez@students.uu.nl

Wouter Swierstra
Utrecht University
w.s.swierstra@uu.nl

Abstract

Datatype generic programming enables programmers to define functions by induction over the structure of types on which these functions operate. This paper presents a library for datatype generic programming in F#, built on top of the .NET reflection mechanism. The generic functions defined using this library can be called by any other language running on the .NET platform.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language constructs and features*]

Keywords datatype generic programming, reflection, F#

1. Introduction

Over the last decade, datatype generic programming has emerged as a powerful mechanism for exploiting *type structure* to define families of functions. In Haskell alone, there are numerous tools and libraries for datatype generic programming, including PolyP (Jansson and Jeuring 1997), Generic Haskell (Hinze and Jeuring 2003), Scrap your boilerplate (Lämmel and Peyton Jones 2003), RepLib (Weirich 2006), Uniplate (Mitchell and Runciman 2007), Regular (Noort et al. 2008), Multi-Rec (Yakushev et al. 2009), Instant Generics (Magalhães and Jeuring 2011) and many others.

Many of these libraries are implemented in the same fashion. They define a *representation type* or *universe* that can be used to describe some collection of types. *Generic* functions are then defined by induction on the elements of the representation type. Finally, these libraries typically support some form of conversion between values of algebraic datatypes and their corresponding representation. This enables users to call generic functions on custom datatypes, without having to implement the underlying conversions manually.

Yet this approach has not been adopted in other languages. In this paper, we will attempt to address this by implementing a library for data type generic programming in F# (Syme et al. 2012b). More specifically, we make the following contributions:

- The type system of F# may not be as rich as that of Haskell, but there are numerous features, including reflection, subtyping, type providers, and active patterns that may be used for type-level programming in F#. We will briefly present the F# features our library relies upon before describing its implementation (Section 2).
- The core of our library is a representation type defined using the sums-of-products adopted by systems such as Generic Haskell (Hinze and Jeuring 2003) and Regular (Noort et al. 2008). We show how such a representation type may be implemented in F# and how the conversion functions relating the values of a datatype to its generic representation may be generated automatically (Section 3).
- Next, we show how generic functions may be defined over this representation type (Section 4). As an example, we will implement a generic map function. Instead of recursing over the representation type directly, we develop several auxiliary functions to hide the usage of .NET reflection and facilitate the definition of generic functions.
- Where many Haskell libraries use type classes to implement type-based dispatch, F#'s overloading mechanism is too limited for our purposes. To address this, we will implement our own system of ad-hoc polymorphism using .NET's reflection. (Section 5)
- Finally, we will show how functions from other Haskell libraries, such as Uniplate, may be readily implemented using the resulting library (Section 6).

We hope that porting the ideas from the datatype generic programming community to F# will pave the way for the wider adoption of these ideas in other .NET languages, such as C#.

WGP '15, August 30, 2015, Vancouver, British Columbia, Canada.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Background

This section introduces the F# language and the .NET platform. Inspired by the 'Scrap your boilerplate' approach to datatype generic programming (Lämmel and Peyton Jones 2003), we will define a company type and a function called *IncreaseSalary*. The function increases the salary of every employee in the company. Our example is different since our library doesn't support mutually recursive data types and we will use this example to present the different type declarations allowed in F#. We will provide an alternative definition of *IncreaseSalary* using our library for generic programming in the second half of this paper.

2.1 The F# language

The F# (Syme et al. 2012b) programming language is a functional language of the ML family. It aims to combine the advantages of functional programming and Microsoft's .NET platform. To do so, the F# designers have adopted numerous features from languages such as Haskell or OCaml, without sacrificing the ability to interface well with other .NET languages. As a result, the language has a much simpler type system than the type system of Haskell, OCaml or Scala. On the other hand, F# performs no type erasure when compiled to the .NET platform.

Before we can define the *IncreaseSalary* function, we will define the types on which it operates:

```
[⟨AbstractClass⟩]
type Employee () = class
    abstract Salary : float with get and set
    abstract NetSalary : float with get
  end
type Metadata =
  { name : string; country : string }
type Staff ⟨'t when 't ≺ Employee⟩ =
  | Empty
  | Member of 't * Staff ⟨'t⟩
type Department⟨'t when 't ≺ Employee⟩ =
  | Tech of Metadata * Staff ⟨'t⟩
  | HR of Metadata * Staff ⟨'t⟩
type Company⟨'t when 't ≺ Employee⟩ =
  | Empty
  | Dept of Department⟨'t⟩ * Company⟨'t⟩
type GuatemalanEmployee (salary' : int) =
  class
    inherit Employee ()
    let mutable salary = salary'
    override self.Salary
      with get () = salary
     and set (value) = salary ← value
    override self.NetSalary
      with get () = self.Salary / 1.12
  end
```

This example demonstrates the different type declarations that F# supports. Besides records, such as *Metadata*, F# supports algebraic datatypes (ADTs) that should be familiar to functional programmers. For example, *Company*, *Department* and *Staff* are ADTs. Furthermore, programmers in F# may define classes, such as *Employee* and *GuatemalanEmployee*. There are several important differences between classes and datatypes. Records and datatypes may be deconstructed through pattern matching and are immutable. In contrast to classes, there is no possible subtyping relation between datatypes. In .NET terminology, they are *sealed*. Classes in F#, on the other hand, behave just as in any other object oriented language. They can inherit from other classes – in our example the class *GuatemalanEmployee* inherits from the *Employee* class. Both ADTs and classes may contain member functions (or methods) declared with the **member** keyword. Member functions always take the object on which they are invoked as an argument, as witnessed by the *self* identifier before a member function's definition.

These data declarations also use polymorphic types and type constraints. In our example, *Company*, *Department* and *Staff* are parametrized by a single type argument. These type arguments have a type constraint, stating that they must be a subtype of the *Employee* class. The type constraints are declared using the **when** keyword.

It is worth pointing out that type arguments can only be of kind $*$. This is a particularly important limitation in the context of datatype generic programming since many Haskell libraries rely on higher kinds.

Next, we implement the *IncreaseSalary* function. To do so, we will begin by defining an auxiliary function called *UpdateEmployee* that applies its argument function to every *Employee* of the company:

```
type Staff ⟨'t⟩ with
  member self.UpdateEmployee (f) =
    match self. with
    | Empty → Empty
    | Member (m, s) →
      Member (f m, s.UpdateEmployee f)
type Department⟨'t⟩ with
  member self.UpdateEmployee (f) =
    match self. with
    | Tech of meta, staff →
      Tech (meta, staff.UpdateEmployee f)
    | HR of meta, staff →
      HR (meta, staff.UpdateEmployee f)
```

```
type Company⟨'t⟩ with
  member self.UpdateEmployee (f) =
    match self. with
    | Empty → Empty
    | Member d, c →
      Member (
        d.UpdateEmployee f,
        c.UpdateEmployee f)
```

Here we have chosen to *overload* the *UpdateEmployee* function, allowing a function with the same name to be defined for different types. To overload functions in F#, they must be defined as a member function. Member functions may be defined post-hoc, i.e., after the type has been defined.

Using *UpdateEmployee*, the *IncreaseSalary* function can be defined as follows:

```
type Company⟨'t⟩ with
  member self.IncreaseSalary (v) =
    self.UpdateEmployee (
      fun e → e.Salary ← e.Salary + v; e)
```

Note that because we have defined the *Employee* type as a class, it is passed by reference in the *UpdateEmployee* function. The argument function we pass to *UpdateEmployee* mutates the object's *Salary* property directly and subsequently returns the argument object.

In the later sections, the *UpdateEmployee* function will be defined in terms of a generic map implemented with our library. Before doing so, we will give a brief overview of some of the relevant features of the .NET platform.

2.2 The .NET platform

The .NET platform is a common runtime environment supporting a family of languages. It provides languages with a type system that includes support for generics and reflection. Many operations on types in F#, such as casting, are handled by the .NET platform.

Subtyping The .NET platform defines the subtyping relation that is used by F#. We write $\tau_a \prec \tau_b$ to denote that τ_a is a subtype of τ_b. In F#, such subtyping constraints can be specified in a type by writing $\tau_a :> \tau_b$.

In any language running on .NET, it is possible to cast a value to some other (super)type explicitly. When assigning a type τ to a value x, it is necessary to check that x is of type τ. In some cases, this check can be done statically. We write $x \prec \tau$ (written $x :> \tau$ in F#) for statically checked casts. In other cases, this check can only be done dynamically. We write $x \precsim \tau$ (in F# $x :?> \tau$) for dynamically checked casts. Dynamically checked casts are usually necessary when using reflection. If a dynamically checked cast fails, an exception is thrown.

As in most object oriented languages, the .NET subtyping mechanism allows values to be cast implicitly to any supertype. The F# language uses the keyword **inherit** to denote that a type inherits from another type. The subtyping relation does not extend automatically to parametrized types: that is, the implication $\tau_a \prec \tau_b \Rightarrow \mathtt{T}\langle\tau_a\rangle \prec \mathtt{T}\langle\tau_b\rangle$ does not hold in general.

Reflection To handle all type operations and collect type information, the .NET platform defines the abstract class *Type*. Instances of the class *Type* encode all the type information about values. When F# is compiled to the .NET intermediate language, CIL, an instance of the *Type* class is generated for every type defined in the F# code. The .NET platform makes this type information available at runtime. Every object has the method *GetType* which returns the value of type *Type*.

The *Type* class is not sealed. Languages can extend it with any information they want. This allows F# to include specific metadata about algebraic datatypes and other F# specific types in the *Type* class. We can use this metadata to query the constructors of an algebraic datatype, or even to pattern match values with its type constructors. It is also possible to use reflection to invoke the type constructors of an ADT to create values of that type. Using reflection is not type-safe in general since the information gained through reflection is only available at run-time. Any errors will cause a runtime exception. Nevertheless, reflection is actively used in libraries such as FsPickler (Tsarpalis 2013), a general purpose .NET serializer, and Nancy (Håkansson and Robbins), a .NET web framework.

Code written using reflection is usually cluttered with method calls that execute .NET internal routines but have no relevance in the logic of the algorithm it implements. This makes it very inconvenient to use in ordinary code. It also leads to code that is hard to maintain since its use requires good understanding of .NET's internals.

3. Type Representations in F#

The core of most libraries for datatype generic programming is a *representation type* or *universe*. It determines the types that can be represented and how generic functions are defined. We will adopt the sums-of-products view of algebraic datatypes, as pioneered by Generic Haskell (Hinze and Jeuring 2003) and libraries such as Regular (Noort et al. 2008).

The type system of F# is not as expressive as Haskell's type system. In particular, all type variables are necessarily of kind $*$; furthermore, all calls to overloaded methods must be resolved statically and cannot depend on how type variables are instantiated. For these reasons, we will need to adapt the Haskell approach slightly.

We will define an abstract class, *Meta*, that will be used to define type representations. Its purpose is to impose type constraints on type variables. These constraints serve as an alternative to typeclass constraints that are used in Regular. For example, (a slight variation of) the following instance is defined in the Regular library:

```
[⟨AbstractClass⟩]
type Meta () = class end

type U⟨'ty⟩() =
  class
    inherit Meta ()
  end

type K⟨'ty,'x⟩(elem:'x) =
  class
    inherit Meta ()
    member self.Elem
      with get () = elem
  end

type Id⟨'ty⟩(elem:'ty) =
  class
    inherit Meta ()
    self.Elem
      with get () = elem
  end

type Sum⟨'ty,'a,'b
  when 'a ≺ Meta
  and 'b ≺ Meta⟩(
  elem: Choice⟨'a,'b⟩) =
  class
    inherit Meta ()
    member self.Elem
      with get () = elem
  end

type Prod⟨'ty,'a,'b
  when 'a ≺ Meta
  and 'b ≺ Meta⟩(
  e1:'a, e2:'b) =
  class
    inherit Meta ()
    member self.Elem
      with get () = e1, e2
    member self.E1
      with get () = e1
    member self.E2
      with get () = e2
  end
```

Figure 1: Definition in F# of all the types used to build type representations.

```
instance (GMap f, GMap g) ⇒
  GMap (f :*: g) where
    gmap f (x :*: y) = ...
```

Instead of abstracting over higher-kinded type arguments, we will abstract over first-order type variables of kind $*$, and constrain them to be a sub-type of *Meta*.

In the remainder of this section, we will present the concrete subtypes of the *Meta* class defined in our library. All the subclasses of the *Meta* class are parametrized by at least one (phantom) type argument, '*ty*. This argument will be instantiated to the type that a value of type *Meta* is used to represent.

The first subclass of *Meta* is *Sum*, that represents the sum of two types. Besides the type argument '*ty*, the *Sum* type constructor takes two additional type arguments: '*a* and '*b*. The *Sum* class stores a single piece of data, namely a value *elem* of type *Choice*⟨'*a*, '*b*⟩. The *Choice* type in F# is analogous to Haskell's *Either* type. It has two constructors: *Choice1Of2* and *Choice2Of2*. Note that both '*a* and '*b* have the constraint that '*a* and '*b* are subtypes of the *Meta* class.

The second subclass of *Meta* is *Prod*, corresponding to the product of two types. Besides the '*ty* argument, the *Prod* type accepts two additional type arguments: '*a* and '*b*. Once again, we require both '*a* and '*b* to be subtypes of the *Meta* class. Besides products, we will use the class $U \prec Meta$ to represent the unit type which takes no extra type arguments.

Next, the subclass *K* of *Meta* is used to represent types not defined as algebraic datatypes. This can be used to represent primitive types, such as *int* or *float*. The *K* constructor takes one extra type argument, '*a*, corresponding to the type of the value it stores. Since F# cannot statically constrain a type to be an algebraic datatype or not, '*a* has no constraints.

Finally, *Id* is the last subclass of *Meta*. This type is used to represent recursive types. It only takes the '*ty* type argument used to refer to recursive occurrences of values of the type being represented.

The definitions of these types are given in Figure 1. These definitions are not complete since the actual implementation contains extra code used for reflection which is not relevant when discussing the universe of types that our library can handle. The full definition can be found in the source code (Rodriguez 2015).

We conclude this section with an example of our type representation. Given the following ADT in F#:

```
type Elems = Cons of int * Elems
           | Val of int
           | Nil
```

We can represent this type as a subtype of the *Meta* class as follows:

```
type ElemsRep =
  Sum⟨
    Elems,
    Sum⟨
      Elems,
      Prod⟨Elems, K⟨Elems, int⟩,
        Prod⟨Id⟨Elems⟩⟩, U⟨Elems⟩⟩⟩⟩,
      Sum⟨
        Elems,
```

$$Prod\langle K\langle Elems\rangle, int\rangle, U\langle Elems\rangle\rangle\rangle,$$
$$U\langle Elems\rangle\rangle\rangle,$$
$$U\langle Elems\rangle\rangle\rangle$$

This example shows how nested Sum types are used to represent the three type constructors of the $Elems$ type. We show how constructors of different number of arguments are encoded with the $Prod$ type. The recursive occurrence of $Elems$ in the $Cons$ constructor is represented by the Id type.

There is some overhead in this representation – we could simplify the definition a bit by removing spurious unit types. However, it is important to emphasize that these definitions will be *generated* using .NET's reflection mechanism. To keep the generation process as simple as possible, we have chosen not to optimize the representation types.

Generating conversion functions

The representations are used as a universal interface to implement algorithms that work on a family of types. It is important to perform the conversion between types and their representations automatically; otherwise the cost of using a library for generic programming may exceed writing instances of generic functions by hand. Haskell libraries usually use Template Haskell (Sheard and Peyton Jones 2002) to generate these conversions. Some Haskell compilers even have a built-in mechanism (Magalhães et al. 2010) for these conversions. The F# language does not have the same facilities for meta-programming but we can use .NET's reflection mechanism to achieve similar results.

The $Object$ class of .NET has a method called $GetType$: $unit \to Type$ which returns a value that contains all the information about the type of the object on which it is invoked. Since $Type$ is an abstract class, every language hosted on the .NET platform is free to extend the precise information stored in instances of the $Type$ class. This allows the F# compiler to include metadata that can be used to query the (type of the) constructors of any algebraic data type at runtime.

Using this $GetType$ member function, we can obtain type information at runtime which is used to traverse the $Type$ value and generate the necessary conversion functions. This functionality is implemented by the $Generic\langle `t\rangle$ class with the following members:

$$\begin{array}{l}\textbf{type}\ Generic\langle `t\rangle() = \\ \quad \textbf{member}\ x.To : `t \to Meta \\ \quad \textbf{member}\ x.From : Meta \to `t\end{array}$$

Note that these conversions are generated *dynamically*, in contrast to most Haskell approaches to generic programming. Intermediate results of this conversion can be cached to reduce the performance penalty.

4. Generic functions

The purpose of type representations is to provide an interface that the programmer can use to define generic functions. Once a function is defined on all the subtypes of the $Meta$ class, it can be executed on any value whose type may be modeled using the $Meta$ class.

To illustrate how generic functions may be defined, we will define a generic map operator, $gmap$. This function accepts as an argument a function of type $\tau \to \tau$ and applies the function to every value of type τ in a ADT. In Regular, a generic function is defined as a typeclass. In our library, we define $GMap$ as a .NET class. Every generic function in our library is implemented as a subclass of the $FoldMeta$ class. This is an abstract class that specifies the minimal implementation required to define a generic function. Its definition is given in Figure 2.

$$\begin{array}{l}\textbf{type}\ GMap\langle `t, `x\rangle() = \\ \quad \textbf{class} \\ \quad \textbf{inherit}\ FoldMeta\langle \\ \quad\quad `t, \\ \quad\quad `x \to `x, \\ \quad\quad Meta\rangle() \\ \quad //\ [...]\texttt{Implementation}[...] \\ \quad \textbf{end}\end{array}$$

The $FoldMeta$ class is parametrized by three type arguments: $`t$ which is the type on which the generic functions are invoked, $`in$ which is the input type of the function, $`x \to `x$ in this case, and $`out$ which is the return type of the generic function, in this case $Meta$.

The $FoldMeta$ class specifies the definitions necessary to compute some new result from a value of type $Meta$. Roughly speaking, it requires a method for each concrete subtype of the $Meta$ class. Note that all the methods are universally quantified over the type being represented, $`ty$. The only exception is the $Id$, representing recursive occurrences, that requires a member function of type $Id\langle `t\rangle * `in \to `out$, where $`t$ is the first type argument of $FoldMeta$.

To illustrate why this distinction is necessary, consider invoking the $GMap$ function on the $Company$ type. This type contains values of type $Department$. As a result, intermediate Sum constructors may have a type that is of the form $Sum\langle Department\langle \_\rangle, \_, \_\rangle$ or a type of the form $Sum\langle Company\langle \_\rangle, \_, \_\rangle$. In order to handle both of these cases, the $`ty$ argument of the $Sum$ type must be universally quantified in the corresponding definition of $FoldMeta$. On the other hand, recursive occurrences always store a value of the type being represented. Hence the $FoldMeta$ definition for the $Id$ type constructor is specialized to $`t$ rather than being polymorphic for any $`ty$. Here we adopt the convention that $`t$ always refers to the type being represented and $`ty$ is a universally quantified type variable corresponding to the specific type of each method.

By overriding the $FoldMeta$ methods in the concrete $GMap$ class, we define the desired map operation. The $FoldMeta$ class and its member functions will be explained in detail in Section 5; for the moment, we will restrict our-

```
[⟨AbstractClass⟩]
type FoldMeta⟨'t,'in,'out⟩() =
  abstract FoldMeta: Meta * 'in → 'out
  abstract FoldMeta⟨'ty⟩: Sum⟨'ty, Meta, Meta⟩ * 'in → 'out
  abstract FoldMeta⟨'ty⟩: Prod⟨'ty, Meta, Meta⟩ * 'in → 'out
  abstract FoldMeta⟨'ty,'a⟩: K⟨'ty,'a⟩ * 'in → 'out
  abstract FoldMeta: Id⟨'t⟩ * 'in → 'out
  abstract FoldMeta⟨'ty⟩: U⟨'ty⟩ * 'in → 'out
```

Figure 2: Definition of the $Meta$ abstract class for generic functions taking one argument.

selves to the methods that we override in the $GMap$ class. The first method we override handles the Sum type constructor:

```
override self.FoldMeta⟨'ty⟩
  (v: Sum⟨'ty, Meta, Meta⟩
  ,f:'x → 'x) =
    match v.Elem with
    | Choice1Of2 m →
      Sum⟨'ty, Meta, Meta⟩(
      self.FoldMeta (m,f) |> Choice1Of2)
    | Choice2Of2 m →
      Sum⟨'ty, Meta, Meta⟩(
      self.FoldMeta (m,f) |> Choice2Of2)
    ≺ Meta
```

This example uses the following F# specific constructs:

- the the pipeline operator ($|>$) which is simply reversed function application: $x \mid> f = f\ x$. This is a common operator used in F#, analogous to the ($) in Haskell but asociates to the left and has its arguments flipped.
- The **override** keyword serves the same purpose as **member** but results in a compile time error if no matching **member** is found in the super-class.

The definition is fairly unremarkable: it pattern matches on its argument and applies the $FoldMeta$ function to the values contained in the Sum type. It then reconstructs a Sum instance with the results of the recursive call. Lastly, it casts the result back to the type $Meta$. The recursive calls of $FoldMeta$ happen through the member function $FoldMeta: Meta * 'in \rightarrow 'out$ of the $FoldMeta$ class. We defer its description to section 5. The next definition handles products:

```
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
```

The type $Prod$ contains the properties $E1$ and $E2$ which store the two constituent elements of the product. Once again, we recursively invoke $FoldMeta$ on these values, reassemble the result and cast it back to the type $Meta$. The definition for unit types, U, is similarly unremarkable.

We define two cases to handle the K type constructor:

```
member x.FoldMeta⟨'ty⟩(
  v: K⟨'ty,'x⟩, f:'x → 'x) =
  K (f v.Elem) ≺ Meta
override x.FoldMeta⟨'ty,'a⟩(k: K⟨'ty,'a⟩
  ,f:'x → 'x) = k ≺ Meta
```

The first definition defines a new member function. It applies the function f to a value of type $'x$. The property $Elem$ of the K constructor returns the value of type $'x$, which we pass to f, before casting the result back to a value of type $Meta$. The second definition overrides the required $FoldMeta$ member function on K; this definition leaves the underlying value untouched. The case for the Id constructor is a bit more

involved:

```
override x.FoldMeta (v:Id⟨'t⟩, f:'x → 'x) =
  let g = Generic⟨'t⟩()
  Id⟨'t⟩(x.FoldMeta (
    g.To c.Elem, f) |> g.From)
  ≺ Meta
```

The *Id* case of the abstract *FoldMeta* member instantiates the $'ty$ argument of the *Id* constructor to $'t$. This means that the *Id* case only needs to be specified for the type $'t$, the type to which the generic function is being applied. Contrary to the other overloads, it is not universally quantified for all types. The *Id* constructor stores a single value, *Elem*, of type $'t$. Using the $Generic\langle'ty\rangle$ class it is possible to convert this $'t$ to a value of type *Meta*. After calling the *FoldMeta* function recursively, we can convert the result back to the original type.

We have now completed the required definitions for our *GMap* class but there still remains one problem. We have assumed that all algebraic data types will be converted to a value of type *Meta*, after which we may apply overridden methods to obtain the desired result. Now suppose the *function* we are mapping has the type $X \to X$, for some algebraic data type X. In that case, we do *not* want to convert values of type X to their corresponding representation (and apply the generic *GMap* function), but rather we would like to transform these values using the argument function. To resolve this, we need to provide several additional definitions.

We will define four additional member functions with a more specific type. Recall that in our declaration of *GMap*, we stated that the argument f has type $'x \to 'x$. Each of the new member functions will specifically work on representations of the type $'x$, that is, the type of values being transformed using the *GMap* function:

```
let mapper (f:'x → 'x) (v:Meta) =
  let g = Generic⟨'x⟩()
  v |> g.From |> f |> g.To
member x.FoldMeta (
  u:U⟨'x⟩, f:'x → 'x) = mapper f u
member x.FoldMeta⟨'a⟩(
  u:K⟨'x,'a⟩, f:'x → 'x) = mapper f u
member x.FoldMeta (
  p:Prod⟨'x, Meta, Meta⟩, f:'x → 'x) = mapper f p
member x.FoldMeta (
  s:Sum⟨'x, Meta, Meta⟩, f:'x → 'x) = mapper f s
```

All of these member functions behave similarly: they convert the generic representation back to a value of type $'x$, apply the function f, and convert the result back to its corresponding representation of type *Meta*. Now we can use the $GMap \prec FoldMeta$ 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
```

Calling this function, requires dispatching on the representation type, which is handled by the $FoldMeta : Meta * $ `'in -> '` $ out$ member function. An instance of *GMap* with $\langle't\rangle$ set to *Company* and $\langle'x\rangle$ set to *Employee* would implement the *UpdateEmployee* function introduced in section 2.1.

5. The *FoldMeta* class

In the previous section, we assumed the existence of a *FoldMeta* function with type $Meta * ('x \to 'x) \to Meta$. Before getting into the details of this function, we would like to revisit the problem that it needs to solve. Consider the following instances that define a fragment of a generic *map* function in Haskell:

```
instance (GMap a, GMap b) ⇒
  GMap (a :+: b) where
  gmap (L a) f = L (gmap a f)
  gmap (R b) f = R (gmap a f)
instance (GMap (K Int)) where
  gmap (K v) f = K (f v)
instance GMap U where
  gmap U _ = U
```

Let's take a look at the *GMap* definition for the :+: type. This function makes a recursive call to *gmap* – but which overload will get called? There are three different overloads to choose from. In Haskell, a choice is not made until enough type information is present. The *GMap* function might be invoked with a value of type $U :+: U$, or a value of type $K\ Int :+: K\ Int$, or even $GMap\ a \Rightarrow a :+: K\ Int$. The chosen overload depends on the types at the *call site* and might be different in every call.

We could try adopting a similar approach in F#, by defining the following member functions:

```
member x.FoldMeta⟨'ty,'a,'b⟩(
  s:Sum⟨'ty,'a,'b⟩, f:int → int) =
  match s.Elem with
  | Choice1Of2 a →
    Sum⟨'ty,'a,'b⟩(
      x.FoldMeta (a, f) |> Choice1Of2)
  | Choice2Of2 b →
    Sum⟨'ty,'a,'b⟩(
      x.FoldMeta (b, f) |> Choice1Of2)
```

```
member x.FoldMeta⟨'ty⟩(
  k:K⟨'ty, int⟩, f:int → int) = (K (f k.Elem))
member x.FoldMeta⟨'ty⟩(
  u:U⟨'ty⟩, f:int → int) = u
```

However, this code is rejected by the F# compiler. At the definition site of the *Sum* case of *FoldMeta*, it is still unclear how to resolve the recursive calls to specific overloads. The F# compiler cannot wait until *FoldMeta* is applied to a value and more type information is present in order to decide which function to invoke. In object oriented languages, abstract methods are usually used to dispatch different methods depending on the argument. This approach does not work for our purposes because it would require that every specific instance of a polymorphic type overrides the method differently. For example, if for the K constructor one wanted a *FoldMeta* specialized for *int* and another polymorphic for all $'t$; $K\langle int\rangle$ and $K\langle't\rangle$ would require a different override of *FoldMeta*.

We resolved this problem by defining a *FoldMeta* function of type $Meta * 'in \to 'out$. This function can also be invoked with the internal elements of *Sum* or *Prod* constructors since they are parametrized by variables $'a, 'b \prec Meta$. This *FoldMeta* function then selects the corresponding 'instance' that should be invoked based on the type of its argument. Note that this is handled statically in Haskell, but must necessarily be done dynamically in F#.

To define the *FoldMeta* function that dispatches based on its argument's type, we once again use the .NET reflection mechanism. The *FoldMeta* function inspects the type of its argument. If we have exactly the right method at our disposal, we invoke that method. We only instantiate a more generic method when necessary. This ensures the desired behavior for the two definitions of *GMap* for K that we saw previously, or the use of the *mapper* function to prevent the unfolding of algebraic data types to their representation.

Note that the definition of a new generic function does not require any casting or reflection. That functionality is abstracted away by using a common representation, *Meta*, and a general purpose traversal of such representations, *FoldMeta*.

6. Case study: Uniplate

To further demonstrate how generic functions may be defined using our library, we will implement the *uniplate* function from the Haskell library with the same name. In Haskell, the *uniplate* function has the following type signature:

$$uniplate : Uniplate\ a \Rightarrow a \to ([a], [a] \to a)$$

Given an argument of type a, the *uniplate* function returns a list of all the immediate child nodes of type a and a function that can be used to reassemble the original value, given a list of child nodes. The F# version of *uniplate* should work as follows:

```
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(Val5); Val8]
printf "%A" c
// prints Op("add", Val1, Val2)
printf "%A" (f [Val 1; Val 2])
```

To define the function, we will define two auxiliary generic functions. The first is *Collect* which computes the list of child nodes:

```
type Collect⟨'t⟩() =
  inherit FoldMeta⟨'t, 't list⟩()
  member self.FoldMeta⟨'ty⟩(
    c:Sum⟨'ty, Meta, Meta⟩) =
    match c.Elem with
    | Choice1Of2 m → self.Collect m
    | Choice2Of2 m → self.Collect m
  override self.FoldMeta⟨'ty⟩(
    c:Prod⟨'ty, Meta, Meta⟩) =
    List.concat⟨'t⟩[
      self.Collect c.E1
      ; self.Collect c.E2]
  override self.FoldMeta⟨'ty, 'a⟩(
    _:K⟨'ty, 'a⟩) = []
  override self.FoldMeta⟨'ty⟩(_:U⟨'ty⟩) = []
  override self.FoldMeta (i:Id⟨'t⟩) =
    [i.Elem]
```

The function is straightforward to understand. Values of the *Sum* type are processed recursively; the results of products are combined by concatenating the resulting lists. Constants and unit types return an empty list. The only interesting case is that for the *Id* type constructor, which returns a singleton list with its associated value. Note that the *FoldMeta* class only has two type arguments ($'t$ and $'t\ list$), in contrast to *GMap* that had three type arguments. To allow generic functions with a variety of arguments, our library defines several variations of the *FoldMeta* class. F# allows types with the same name and different number of type arguments to coexist in the same namespace.

The second generic function we define is *Instantiate*, which reconstructs the value of an algebraic datatype when passed the list of child nodes. We will store this list in a local, mutable variable *value*, to which each of the instance definitions below may refer.

```
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
...
```

To complete this definition, we need to define suitable instances for the subclasses of $Meta$. The most interesting case is that for the Id type constructor:

```
override self.FoldMeta (i:Id⟨'t⟩) =
  match pop () with
  | Some x → Id⟨'t⟩(x)
  | None → failwith "Not enough args"
  ≺ Meta
```

To produce the desired child, we *pop* it off the mutable list of children we have at our disposal. If such child doesn't exist, the list we passed is too short and the function fails.

The case of sums and products are analogous to the $Collect$ function, making two recursive calls to construct a new $Meta$ value:

```
override self.FoldMeta⟨'ty⟩(
  p:Prod⟨'ty, Meta, Meta⟩) =
  Prod (self.FoldMeta p.E1, self.FoldMeta p.E2)
  ≺ Meta
member 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
```

Note that these definitions rely on the list of values being mutable and F#'s call-by-value semantics. In the case for products, we know that the first call $self.FoldMeta\ p.E1$ will be evaluated first, consuming the required child nodes from the list of values, before evaluation continues with the second component of the tuple.

Finally, the cases for the type constructors U and K are trivial, as they do not need to modify the list of $values$.

```
override self.FoldMeta⟨'ty⟩(u:U⟨'ty⟩) =
  u ≺ Meta
override self.FoldMeta⟨'ty,'a⟩(k:K⟨'ty,'a⟩) =
  k ≺ Meta
```

The $uniplate$ function simply wraps both these results together and handles the conversions to our type representation:

```
let uniplate⟨'t⟩(x:'t) =
  let g = Generic⟨'t⟩()
  let rep = g.To x
  let xs = rep |> Collect⟨'t⟩().FoldMeta
  let inst xs' =
    rep |> Instantiate⟨'t⟩(xs').FoldMeta⟨'t⟩
      |> g.From
  (xs, inst)
```

7. Limitations of the $FoldMeta$ class

The $FoldMeta$ class provides a generic interface that abstract away the use of reflection to traverse and construct values of many types generically. Requiring generic functions to be defined using the $FoldMeta$ class is more restrictive than the approach that many Haskell libraries use. For example, it is not obvious how to define a generic equality function, since that requires induction on two values instead of one.

Variants of the $FoldMeta$ class that perform induction over more than one value are possible. We only need to enforce that all cases for one argument are covered to ensure that pattern matching is exhaustive. For example, we can extend $FoldMeta$ as:

```
[⟨AbstractClass⟩]
type FoldMeta⟨'t,'out⟩() =
abstract FoldMeta
  :Meta * Meta → 'out
abstract FoldMeta⟨'ty⟩
  :Sum⟨'ty, Meta, Meta⟩ * Meta → 'out
abstract FoldMeta⟨'ty⟩
  :Prod⟨'ty, Meta, Meta⟩ * Meta → 'out
abstract FoldMeta⟨'ty,'a⟩
  :K⟨'ty,'a⟩ * Meta → 'out
abstract FoldMeta
  :Id⟨'t⟩ * Meta → 'out
abstract FoldMeta⟨'ty⟩
  :U⟨'ty⟩ * Meta → 'out
```

The programmer can then overload any of the methods with specific instances of the second argument which can then be pattern matched with reflection. For example, to define generic equality, one would like to have an overload of the Sum case with type:

```
member FoldMeta⟨'ty⟩
  :Sum⟨'ty, Meta, Meta⟩ * Sum⟨'ty, Meta, Meta⟩
  → 'out
```

If $FoldMeta$ were invoked with both arguments being Sum, then this overload would get called. If the second argument is not Sum, then the overload accepting a $Meta$ would be invoked. In this way, it is possible increase the class of generic functions definable by our library.

Another limitation of the current implementation is that all the overloads of *FoldMeta* must return a value of the same type. More advanced libraries for datatype generic programming use some limited form of dependent types, possibly through type classes or type families, to allow generic functions to return values of different types. The *FoldMeta* class lacks such mechanism as it can be used to subvert the F# type system. Consider the following example:

```
member self.FoldMeta⟨'ty⟩(
  v:K⟨'ty, Employee⟩) = K (v.Elem) ≺ Meta
```

The type checker would not object to changing the function as follows:

```
member self.FoldMeta⟨'ty⟩(
  v:K⟨'ty, Employee⟩) =
  K ("I am not an Employee!!") ≺ Meta
```

This function now changes the type of the value stored in the K constructor, before casting it to the *Meta* type. This is type correct since any instance of K is a subtype of *Meta*. However, if the result of this function is passed to the *From* function that generated the original representation it results in a runtime error.

Such errors could be prevented by revisiting the previous definition of the *FoldMeta* class and adding additional type parameters for each required overload.

```
type FoldMeta⟨
  't,  // Generic type
  'm, // Return type of the Meta overload
  's,  // Return type of the Sum overload
  'p,  // Return type of the Prod overload
  'i,  // Return type of the Id overload
  'k,  // Return type of the K overload
  'u,  // Return type of the U overload
  ⟩
```

However, to perform recursive calls, all overloaded functions invoke the overload specialized for the *Meta* type, which dispatches as discussed in section 5. Since the current implementation requires all the overloaded definitions to have the same type, the method does not need to check that the return type of the overload it selects is correct. This would no longer be the case if different return types are permitted. The dispatch could fail at runtime if the selected overload returns a different type. The problem can be solved by enforcing that all overloads return values which are a subtype of some other type, in this case $'m$, so the dispatcher can safely cast the result to this type. This can be enforced with additional type constraints:

```
type FoldMeta⟨
  // [...]
  when 's ≺ 'm
  and 'p ≺ 'm
  and 'i ≺ 'm
  and 'k ≺ 'm
  and 'u ≺ 'm
  ⟩
```

Unfortunately, type constraints in F# can only be used to enforce that a type must be a subclass of a *concrete* type, not a type variable. One alternative is to make the subtyping relation explicit with the help of member constraints:

```
type FoldMeta⟨
  // [...]
  when 's:(member Cast: unit → 'm)
  and 'p:(member Cast: unit → 'm)
  and 'i:(member Cast: unit → 'm)
  and 'k:(member Cast: unit → 'm)
  and 'u:(member Cast: unit → 'm)
  ⟩
```

These member constraints impose the requirement that a member function of the specified type is present in the type that instantiates the variable. This way the dispatcher *FoldMeta* member can safely cast the result into type $'m$ by calling the *Cast* method of the given value. Although this approach may work in principle, it highlights some of the limitations of F# that we have encountered.

There are a few cases where our usage of subtyping and the *FoldMeta* class has certain advantages compared to many Haskell libraries. Suppose we want to define an alternative version of the *GMap* function, *ShallowGMap*, that does not traverse recursive occurrences of its argument. Essentially, these two definitions are the same – the only difference is in the *Id* case. Using the F# approach we have described, we can define a new subclass of *GMap*, overriding the instance for *Id*:

```
type ShallowGMap⟨'t,'x⟩() =
  inherit GMap⟨'t,'x⟩()
  override self.FoldMeta (i:Id⟨'t⟩, f:'x → 'x) = i
```

This is harder to do in most Haskell libraries. Generic functions defined using type classes, such as those in the Regular library, make it very hard to re-use existing instance definitions in new generic functions. The class and module system of F#, on the other hand, make it fairly straightforward to define different variations of the same function in the same namespace.

8. Discussion

This paper aims to explore the possibility of using of datatype generic programming in F#. To do so, we had to

adapt the existing approaches to match F#'s type system. We also substituted some type level computations performed by the Haskell compiler with reflection. In the remainder of this section, we will look back on our library and the limitations of F#.

First of all, due to the use of reflection, our representation types are defined as classes, instead of (generalized) algebraic data types or type classes as typically done in most Haskell libraries. As a result, type-class constraints, as used by Regular, were translated to subtype constraints in F#. Due to the limitations of F#, we manually implemented part of the type-directed dispatch of Haskell's type classes in our own $FoldMeta$ class. As a result, we need to do more work at runtime, even if memoizing results can help improve performance.

The library makes it possible to define a wide variety of generic functions. Our implementation of the $uniplate$ method highlights that new generic functions can be added without having to use any .NET reflection. The expressiveness of the library is limited by the fact that the programmer must use the $FoldMeta$ class as the interface to define generic functions. This is done to ensure complete definitions. It is possible to provide variants of $FoldMeta$ to increase expressiveness, but this clutters the library with definitions. An alternative would be to allow the programmers to provide their own definitions of the $FoldMeta$ class. This could be achieved by requiring that the programmer annotates the definition with a special pragma. Then the compiled F# assemblies can be checked post-compilation for consistency using reflection since all the type information is still present after compilation and the compiled assemblies can be dynamically loaded. This technique is used by Websharper (Granicz 2015) to compile F# into Javascript.

One of the advantages of using representations to define generic functions is the ability to construct values generically. Although our library allows it, it is much less convenient than using Haskell since our use of subtyping and casting values to the $Meta$ class might result in unsafe *generic* functions. That is, a type-correct generic function may return a result that throws an exception once it is converted back to an algebraic data type. We already saw this in the definition of $GMap$ – it was certainly possible to produce ill-formed representation types and cast the result back to the $Meta$ type. The same problems can also occur when defining other generic functions, such as $read$, that need to produce a representation of type $Meta$. The F# type system will not object to definitions that produce the wrong kind of representation. It might be possible to increase safety by implementing additional checks using reflection and this is left as future research.

There are alternative mechanisms that could be used for datatype generic programming in F#. Type providers (Syme et al. 2012a) can be used to generate types at compilation time by executing arbitrary .NET code. They are typically used to provide typed access to external data sources, such as a database. Although we have experimented with using type providers to generate representation types, we abandoned this approach. Type providers have several severe restrictions on the type information they can access and the types they can generate. Alternatively, the *quotations* library (Syme 2006) provides constructs to generate F# programs at run-time. Even though some functions can be implemented generically by using quotations, the approach has no convenient way to pattern match values generically.

Datatype generic programming has been attempted in other object oriented languages such as Scala (d. S. Oliveira and Gibbons 2010). This library makes makes use of higher-kinds and traits to define representations and generic functions are defined by cases in a similar fashion as in Regular. The library uses implicit parameters instead of reflection for method dispatch. Type representations must be specified manually but the compiler can typecheck the correctness of the definition. The approach also enjoys the extensibility advantages present in our library.

The idea of datatype generic programming is now almost twenty years old (Jansson and Jeuring 1997; Backhouse et al. 1999). Yet the approach has not been widely adopted outside of the functional programming community, despite several attempts to implement libraries in languages such as Scala (d. S. Oliveira and Gibbons 2010). We believe that there is still much work to be done to transfer expertise from the datatype generic programming community to other, more mainstream programming languages. We hope that the library we have presented here is another small step down that road.

Acknowledgments

We would like to thank the Software Technology Reading Club of the University of Utrecht and our reviewers for their helpful feedback.

References

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming. In *Advanced Functional Programming*, pages 28–115. Springer, 1999.

B. C. d. S. Oliveira and J. Gibbons. Scala for generic programmers. *Journal of Functional Programming*, 20(3,4):303–352, 2010. . URL `http://www.comlab.ox.ac.uk/ jeremy.gibbons/ publications/ scalagp-jfp.pdf`. Revised version of the WGP2008 paper.

A. Granicz. Functional web and mobile development in f#. In V. Zsk, Z. Horvth, and L. Csat, editors, *Central European Functional Programming School*, volume 8606 of *Lecture Notes in Computer Science*, pages 381–406. Springer International Publishing, 2015. ISBN 978-3-319-15939-3. . URL `http://dx.doi.org/10.1007/978-3-319-15940-9_9`.

R. Hinze and J. Jeuring. Generic Haskell: Practice and Theory. In *Generic Programming*, pages 1–56. Springer, 2003.

A. Håkansson and S. Robbins. Nancy. `http://nancyfx.org`.

P. Jansson and J. Jeuring. Polyp – a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM, 1997.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM. ISBN 1-58113-649-8. . URL `http://doi.acm.org/10.1145/604174.604179`.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Symposium on Haskell*, Haskell '10, pages 37–48, 2010. ISBN 978-1-4503-0252-4. .

J. P. Magalhães and J. Jeuring. Generic programming for indexed datatypes. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 37–46. ACM, 2011.

N. Mitchell and C. 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. ISBN 978-1-59593-674-5. . URL `http://doi.acm.org/10.1145/1291201.1291208`.

T. V. Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *International Conference on Functional Programming*, pages 13–24, 2008. .

E. Rodriguez. A library for datatype generic programming in F#. `http://github.com/netogallo/FSharp-Generics`, 2015.

T. Sheard and S. Peyton Jones. Template meta-programming for haskell. In *In Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 1–16. ACM, 2002.

D. Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*. ACM, 2006. URL `http://research.microsoft.com/ apps/ pubs/ default.aspx? id=147193`.

D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F#3.0 - strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, September 2012a. URL `http://research.microsoft.com/ apps/ pubs/ default.aspx?id=173076`.

D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Apress, November 2012b. URL `http://research.microsoft.com/ apps/ pubs/ default.aspx?id=192596`.

E. Tsarpalis. FsPickler. `http://nessos.github.io/ FsPickler`, 2013.

S. 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. ISBN 1-59593-489-8. . URL `http://doi.acm.org/10.1145/1159842.1159844`.

A. R. Yakushev, S. Holdermans, A. Lh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*, pages 233–244, 2009. .