Generic Programming in F#

Ernesto Rodriguez
Student: 4083369

Computer Science
Utrecht University
Utrecht
The Netherlands

Type: Master's Thesis
Date: August 31st, 2015
Supervisor: Dr. Wouter Swierstra

Abstract

Datatype generic programming is a programming model that exploits the structural similarities of different types in order to generically define functions on families of types. This model demands a powerful type system so it has been seldomly used outside Haskell. This thesis attempts to address this problem by introducing a datatype generic programming for the F# language, which is a simpler functional language designed for existing .NET programmers.

1 Introduction

Functional programming languages have relied on algebraic data types (ADTs) as the mechanism to represent data structures. They allow inductively defined types which can be de-constructed using pattern matching. Whenever a value is pattern matched inside a function, the function will demand a concrete type (instead of a type variable) for that value. This is very useful since it allows the compiler to infer types but has consequences for certain functions. For example, the equality function is trivial to define. Simply check that both arguments were constructed with the same constructor, if so, apply equality to the arguments of the constructor. However, most languages will require pattern matching to be done in isolation for each type making it impossible to define equality that works on more than one type.

To mitigate the problem, polytypic programming [3], which later became datatype generic programming, was developed. The idea behind polytypic programming is to define functions by induction over the structure of types. The structure is encoded using a representation type and functions are defined on values of the representation type. Finally, a translation between types and representations allows functions to be applied to ordinary values.

Datatype generic programming has been actively researched in the Haskell programming language. Many approaches exists such as Regular [8], Multirec [13], Generic Haskell [5], RepLib [12] and Instant Generics [1]. Most of the approaches differ in the class of types that can be represented by the library – called the universe. In general, if the universe is smaller, the library is easier to learn and its implementation is less demanding for the type system.

Unfortunately, little work has been done to bring these ideas into other programming languages. One of the main difficulties is that most approaches rely on advanced type system features to ensure correct behavior. For example none of the libraries mentioned above works with plain Haskell 98 and all of them use higher kinded polymorphic types. Since most ordinary programming languages still lack these features, the ideas cannot be directly implemented in such languages and need to be adapted.

The present thesis investigates how to adapt the ideas of datatype generic programming to apply them in the F# programming language. The approach is inspired by Regular which is a library designed with ease of use in mind. It leverages from .NET's reflection to carry out the type level computations necessary for generic programming at runtime. To adapt the ideas, several compromises had to be made which resulted in both advantages and disadvantages. The result is packed as a library which can be used to declare generic functions which can be used with little programming overhead in the F# language.

Part I
Background

2 Datatype Generic Programming

Functional programming languages often use algebraic data types (ADTs) to represent values. ADTs are defined in cases by providing a constructor for each case and specifying the type of the values the constructor needs to create a new value. In other words, a type constructor is a function that takes a group of values of different types and produces a value of the ADT's type.

To define functions for ADTs, functional languages provide a mechanism to deconstruct ADTs called pattern matching. This mechanism allows the programmer to check if a particular value was constructed using the specified constructor and extract the arguments used to produce the value. This mechanism is very elegant since it allows defining functions by induction but it has several shortcomings.

A function that pattern matches a value over the constructors of a particular ADT constraints the type of that value to be the ADT defined by those constructors. This leads to functions being implemented multiple times – either when a existing ADT is modified or a new ADT is declared [3]. Due to the importance of abstraction, several methods for polymorphism have been developed to address these restrictions.

An ADT can be higher-kinded. This means that a definition of a list **data** *List* = *Cons Int List* | *Nil* can abstract the type of its content and become **data** *List a* = *Cons a* (*List a*) | *Nil*. A function, such as length, might de-construct the list without performing any operations on its content (the type represented by a). Such function can operate on a list of any type – this is called parametric polymorphism. The programmer might also wish to implement functions that operate on the contents of a list without restricting the type of the list's content to a particular type. This can be done by requiring that the function is also provided with a set of operations that it may perform on its content. For example, the *sum* function could be implemented by requiring that a function to add two elements of type a is provided. This is called ad-hoc polymorphism.

These approaches can be used to define many polytypic functions generically. This is evidenced by the libraries Scrap your Boilerplate Code [4] and Uniplate [7]. Both libraries specify a family of operations that must be supported by a type and use ad-hoc polymorphism to implement many polytipic functions (for example *length* or *increment*) in terms of the family of operations. The programmer only needs to do pattern matching when defining these base operations and both libraries provide mechanisms to do it automatically.

Although it's possible to define many polytipic functions with these approaches, there exists a more general approach called *datatype generic programming.* The following example shows the intuition. For simplicity consider types with constructors that either accept zero or one argument. Such as the types:

```
data Maybe a = Nothing | Just a
data Circle = Radius Int
```

Then typeclasses to match constructors with zero and one argument are defined as follows:

```
class ZeroArgs t where
  value0 :: t → Maybe ()
  constr0 :: t
class OneArg t a where
  value1 :: t → Maybe a
  constr1 :: a → t
```

The following instances are valid for the types given above:

```
instance ZeroArgs (Maybe a) a
instance OneArg (Maybe a) a
instance OneArg Circle Int
```

With those constructors one can easily define an increment function:

```
increment :: OneArg t Int ⇒ t → t
increment v = case value1 v of
  Nothing → v
  Just i → constr1 (i + 1)
```

The idea is encoding type constructors using other types, *ZeroArgs* and *OneArg* in this case, and the arguments of the constructor as type parameters of the encoding type. The type used to encode other types is called *representation type* or *universe*. The approach above has many limitations in the types it is able to encode. For example, it cannot encode a type, such as **data** *Int* = *Pos Nat* | *Neg Nat*, with two type constructors that accept the same argument. Richer representation types are able to encode more types. The remainder of this section introduces Regular [8] which is an approach to *datatype generic programming* able to express many more types. In the rest of this thesis, *generic programming* will always refer to *datatype generic programming* and functions defined using *generic programming* will be called *generic functions*.

2.1 Generic Programming with Regular

Generic functions are defined by induction on the structure of a type. Since pattern matching can only do induction on one type, generic programming libraries include facilities to define functions that work on many types. The most important things are: a *representation type* (U) that can represent values of other types, and a decoding function, $Set \to U$, that decodes representations back to types. The *representation type* along with the decoding function is called the *universe*. The *universe* specifies the types that can be represented by the library.

In the case of Regular, its *universe* consists of all ADTs that:

- Are of kind $*$
- Are not mutually recursive

This universe includes many common types like lists, trees and simple DSLs but is smaller than the set of types definable in Haskell 98. To represent its universe, Regular uses the following types:

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

The types have the following roles:

- *K* represents the occurrence of values of primitive types (eg. *Int* or *Bool*)
- *Id* represents recursion on the type being represented (eg. the *List* argument of the *Cons* constructor).
- *Unit* represents a constructor which takes no arguments
- $(f \oplus g)$ represents sum of two representations. This happens when a type has multiple constructors
- $(f \otimes g)$ represents product of two representations. This happens when a constructor takes multiple arguments.

As an example, this list of integers:

```
data List = Cons Int List | Nil
```

is represented by the type:

```
type Rep = (K Int ⊗ Id) ⊕ Unit
```

It is straightforward to see that the sum of constructors gets translated to the $\oplus$ type. The $\otimes$ type is used to join the arguments of the first constructor. One of the arguments is a primitive *Int* represented with *K Int* and the second arguments is a recursive occurrence of the list, represented by *Id*. Finally, the *Nil* constructor is represented by *Unit*.

The types above are used to define *generic functions* but in order to do so, values must be translated to representations. In Regular, the translation is defined by making a type an instance of the *Regular* typeclass. The *Regular* typeclass is defined as follows:

```
class (Functor (PF a)) ⇒ Regular a where
  type PF a :: ∗ → ∗
  from :: a → PF a a
  to :: PF a a → a
```

The constituents of the class are a type called *PF* which is the representation of the argument type and two functions, *to* and *from*, that convert values to representations and representations to values. In the case of list of *Int*, an instance could be the following:

```
instance Regular List where
  type PF List = (K Int ⊗ Id) ⊕ Unit
  from (Cons i l) = Inl (K i ⊗ Id l)
  from Nil = Inr Unit
  to (Inl (K i ⊗ Id l)) = Cons i l
  to (Inr Unit) = Nil
```

This instance declaration is straightforward. In general, instances of the Regular class are straightforward and libraries usually provide an automatic mechanism to generate them. This feature makes the library easy to use.

Generic functions can now be expressed in terms of values of representation types instead of using values of the type itself. A generic function is specified as a typeclass and is implemented by making representations instances of that class. As an example, the generic increment function will be defined. This function increases the value of every integer that occurs in a type by one. It is defined as the following class:

```
class GInc r where
  gInc :: r → r
```

and is implemented as follows:

```
instance GInc (K Int) where
  gInc (K i) = K (i + 1)
instance GInc Unit where
  gInc Unit = Unit
instance GInc Id where
  gInc (Id r) = Id $ from $ gInc $ to r
instance (GInc f, GInc g) ⇒ GInc (f ⊕ g) where
  gInc (Inl f) = Inl $ gInc f
  gInc (Inr g) = Inr $ gInc g
instance (GInc f, GInc g) ⇒ GInc (f ⊗ g) where
  gInc (f ⊗ g) = gInc f ⊗ (gInc g)
instance GInc (K a) where
  gInc x = x
```

This definition is not very interesting. Whenever there is an integer, its value will be increased by one. In the case of products and sums, the function is recursively applied and the result is packed back into the same product or sum. The case for *Id* also applies the function recursively but since it contains a value, not a representation, it must be converted into a representation to apply *gInc* and the result needs to be converted back to the original type. The rest of the cases leave their argument untouched.

What is important about this function is that Haskell's add-hoc polymorphism (type-classes) is used to perform recursion and to provide type safety. For instance, consider the following definition:

```
instance GInc (K Int) where
  gInc (K i) = K "wrong!"
```

This definition does not type-check since *gInc* :: *a* → *a* but *K i* :: *K Int* and *K* `"wrong!"` :: *K String* which would result in *gInc* :: *K Int* → *K String*.

The definition of *GInc* requires the overlapping instances extension (among others) since there is no way to provide a specific case for *Int* and a case for everything but *Int*. Ommiting the *K Int* case would still result in a valid definition and the Haskell compiler is, in principle, allowed to ignore it. Regular relies on the compiler to select the correct instance and is unable to accept custom selection rules.

For convenience, generic functions are usually wrapped around the conversion functions to provide a toplevel function that works on every instance of the Regular typeclass:

$$\begin{aligned}&\mathit{inc} :: \mathit{Regular}\ a \Rightarrow a \rightarrow a\\&\mathit{inc} = \mathit{from} \circ \mathit{gInc} \circ \mathit{to}\end{aligned}$$

This definition of *GInc* is total for Regular's *universe* since any representation can be applied to it. This is not necessary. Consider deleting the instance: **instance** *Regular* $(K\ a)$. *GInc* will still work for any ADT as long as all of its type constructors take only values of type *Int* as arguments.

3 The F# language

The F# [11] programming language is a functional language of the ML family. It started off as a .NET implementation of OCaml but was adapted so it could inter-opearate with other .NET languages. One notable feature of the language is that it obtains its type system from the .NET platform (hence there is no type erasure). In addition, it includes a syntax directed type inference algorithm inspired on the Hindley-Miller system. The remainder of the section explains some of the components of the F# language.

3.1 Types and Type System

The types in the F# language can be divided into two categories. The purely functional structures (value types) and the Object Oriented/Imperative structures (classes). The language is completely object oriented in the sense that every value is an object. In some cases, the compiler will optimize values by un-boxing primitive types (like integers) but this happens transparently depending on how a value is used.

Value types: The value structures are Algebraic Data Types and Records. Both of theese structures are immutable and do not allow inheritance/sub-type relations (they are sealed in .NET terminology). ADTs in F# are very similar to those of a traditional functional language. Constructors are defined by cases along with the arguments the constructor requires to build the type. Records are defined by enumerating the fields of the record along with the type of each field. Records can then be constructed by providing the arguments of each of the Record's fields as a named argument.

Value types can be de-constructed through pattern matching. F# also supports parametric and ad-hoc polymorphism on these types. Parametric polymorphism is implemented in the same way as in other functional languages. For ad-hoc polymorphism, operations are defined on types as member functions. Interfaces and member constraints can be used to constrain a value to support certain operations. The code below shows the syntax for interfaces and member constraints:

```
[⟨Interface⟩]
type Show =
  abstract Show : unit → string
let print1 ⟨‘x : when ‘x ≺ Show⟩ : ‘x → unit
let print2 ⟨‘x : when ‘x : (member  Show : unit → string)⟩ : ‘x → unit
```

This code defines an interface called *Show* that defines the methods a type should have in order to be convertible to strings. The *print1* function requires that the argument ‘x implements that interface. The *print2* function simply requires that ‘x defines a member function called show with signature *unit* → *string*.

Classes and Structures: The other category of types that exists in F# are classes and structures. Both are very similar with slight differences only on the type parameters they support but those details are not relevant for this thesis and will be ignored. These types are the traditional classes that are available in other object oriented languages. They are defined by providing one (or many) constructors, class variables (which can be mutable) and member functions (or methods). F# (and .Net) allow inheritance from a single type. Classes in F# can also implement any number of interfaces.

Since types can inherit from other types, there exists a sub-typeing relation in F#. This thesis uses the notation $\tau_a \prec \tau_b$ to indicate that τ_a inherits from (is a subtype of) τ_b. As with most OO-languages, values are automatically assigned a supertype if necessary. Sometimes it is necessary to explicitly assign a type to a value. The notation $x \prec \tau_a$ is used to indicate a safe cast of x to τ_a – in other words x is assigned the type τ_a and the compiler can check that the value x is compatible with that type. In some situations, the compatibility cannot be checked statically. When this happens, the operation $x \precsim \tau_a$ is used to dynamically check if x is compatible with τ_a and if so assign the type τ_a to x or raise a runtime exception if x does not have type τ_a.

Finally, classes can define internal or nested types. This is how modules internally work. They simply are classes; toplevel definitions become static members and type definitions become nested types.

Polymorphic types: Types in F# can accept type arguments. These are type variables that can be instantiated to any concrete type as long as the concrete type satisfies the constraints given to the argument. A major difference between F# and other functional laungauges is that type variables are restricted to kind $*$. Functions such as the bind ($\gg=$) function in Haskell cannot be implemented in F#. For example:

$$(\gg=) :: Monad\ m \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

cannot be immitated in F# because m is higher kinded (it takes a as argument). One possible approximation in F# could be:

$$(\gg=) : Monad\langle `a\rangle \to (`a \to Monad\langle `b\rangle) \to Monad\langle `b\rangle$$

and have every monad in F# implement the Monad interface. Even though this funciton would behave correctly, it can go wrong if the first argument is an instance of the *Maybe* monad and the second argument a function that goes from $`a$ to the *IO* monad. Such errors would not be caught by the typechecker.

Member functions: Types are allowed to define member functions (typically known as methods) for any type. Member functions can be abstract or concrete. Abstract members can be overriden by a different implementation with the same signature. They must be overriden at least once in order to instantiate the type. Concrete members cannot be overriden. Member functions can be defined post-hoc in any module. Member functions defined in a different location where the type is defined are called extension members. Whenever a module gets imported, all extension members will be added to their respective types. The major limitation of extension members is that they are not checked when solving memeber constrains.

3.2 Reflection

Through the .NET platform, the F# language has access to a rich reflection library. Reflection is a mechanism that allows programs to query values for information about their type at runtime. In .NET, reflection exposes that information through the *Type* class.

When a program is compiled to CIL, the .NET intermediate language, an instance of the *Type* class is created for every type that is declared in the program. This is an abstract class and specifies all the information that .NET needs about a type. Languages of the .NET platform extend the *Type* class with the information they wish to store about their types. In the case of F#, information about the constructors and record fields is included in the type.

The .NET platform is an object oriented runtime system. Every value is an object and has methods asociated with it. When ADTs get translated to native .NET values, they become ordinary objects in the object oriented sense. The code below outlines how the structure of an ADT would look like if it were defined as a class:

```
[⟨Sealed⟩]
type List⟨‘a⟩ =
  type Cons⟨‘a⟩(a : ‘a, l : List⟨‘a⟩)
    inherit List⟨‘a⟩
    member  Value : ‘a * List⟨‘a⟩
  type Nil⟨‘a⟩() = inherit List⟨‘a⟩
  abstract IsCons : bool
  abstract IsNil : bool
```

The definition is not a valid F# definition since classes cannot inherit from a sealed class but compilers targeting .NET are allowed to generate such definitions. This definition ignores some details but includes the information that is necessary to present the reflection library.

In this example, the *List* type has no constructors and cannot be instantiated. Instead, *List* values are created using the *Cons* type or the *Nil* type. Pattern matching is not a native .NET operation. To de-construct a type into a particular pattern, .NET first checks wether the value is a *Cons* or a *Nil* using the respective members and then it performs an unsafe cast to that type and uses the *Value* member function to obtain the values.

Constructing values of type *List* can be done by invoking the constructor of either the *Nil* type or the *Cons* type with the apropiate arguments.

When this ADT gets compiled to .NET, a value of type *Type* is created for *List*, *Cons* and *Nil*. The *Type* class defines the memeber function *GetNestedTypes* : *unit* → *Type* [ ] which returns all the types that are defined inside the type on which the function is invoked. In the case of *List*, *GetNestedTypes* would return an array containing *Cons* and *Nil*. Furthermore, the *Type* class defines a member called *GetConstructor* : *Type* [ ] → *ConstructorInfo*. This member returns a constructor with input types that match the types provided in the array or *null* in case no constructor matches the arguments.

The *ConstructorInfo* class is a subclass of the *MemberInfo* class. This class contains information about member functions. In particular, it defines the member *Invoke* : *obj* → *obj* [ ] → *obj*. This member takes as first argument the object on which it will be invoked and as second argument the values that the member accepts as arguments. Since every type in .NET inherits from *obj*, any values can be passed to this method. The method produces as result the result of calling the method with the provided arguments. If the *Invoke* method is called with arguments of the wrong type it raises a runtime exception. For completeness, the code below uses reflection to construct a simple list.

```
type List⟨‘a⟩ = Cons‘a * List⟨‘a⟩ | Nil
let listTy = (Nil ≺ List⟨int⟩).GetType ()
let [| consTy; nilTy |] = listTy.GetNestedTypes ()
let cons = consTy.GetConstructor ([| typeof⟨int⟩; listTy |])
let nil = nilTy.GetConstructor ([||])
cons.Invoke (null, [| 1 : obj; nil.Invoke (null, [||]) |])
```

Note that when calling static members or constructors using the *Invoke* method, *null* is given as first argument since they don't use that argument.

The reflection api of .NET is very rich. Many more functions are available; an entire book would be required to explain every detail. This section gives the intuition on how it can be used to achieve the objectives of the thesis. More information is available in Microsoft's documentation [6].

Part II

Research Topic

4 Description of the Problem

Even though datatype generic programming has existed for almost 20 years, it is still uncommon in languages other than Haskell. The method is still quite unknown but if more languages adopt it, it could eventually become a tool to prevent boilerplate code and unecessary refactoring within large software systems. However, it is not trivial to translate the Haskell approach to other languages, especially languages like F# which have simpler type systems.

4.1 Why should F# adopt Datatype Generic Programming

Programmers of the F# language also face the problem of having to define a function multiple times for every ADT. Apart from parametric polymorphism and ad-hoc polymorphism, the language dosen't have a good method to define generic functions.

When polymorphism isn't enough, programmers rely on reflection to define functions generically. There are several reasons why this approach is inconvenient:

- Reflection is quite involved to use. The programmer must learn a lot on how .NET internally handles types and values.
- The F# language offers no syntactic facilites to call functions via reflection. This means that a function call can ammount to several lines of code.
- Reflection relies on dyamic typeing which can lead to runtime errors.
- Different implementations (eg. .NET, Websharper and Mono) handle reflection differently so code might not work in every platform.

It is generally easier and less time consuming implementing a function tens of times before recurring to reflection. The average programmer will hardly find it convenient to use reflection, cluttering the codebase with boilerplate code in the long run. Reflection also lacks the mathematical elegance of inductively defined functions which, combined with the disadvantages above, easily leads to code that is hard to mantain.

Taking as inspiration the existing knowlege about datatype generic programming, it might be possible to develop a library that allows the definition of generic functions without requiring the programmer to use reflection. Even if this library is implemented using reflection, the programmer would enjoy several advantages using it:

- The definition of generic functions will not require reflection
- The code that uses reflection can be small and easy to mantain
- The library can perform optimizations which would have to be done manually when using reflection
- Defining and calling generic functions has little overhead for the programmer since it will be done elegantly, inductively and without the syntactic clutter of code that uses reflection

The existing methods for datatype generic programming cannot be directly implemented in F# since the language lacks features that are heavily used by said methods. The remainder of the section introduces these features and explains their role in datatype generic programming.

4.2 Kind-Polymorphism and Datatype Generic Programming

Polytipic programming was introduced as a mechanism to generically derive folds over any type [3]. This approach visualized the representation of a type as the *functor* of the type. A functor f is a typelevel function that takes as argument a type t and produces a new type $f\ t$. Consider the Regular class:

```
class Regular a where
  PF a :: * → *
  from :: a → PF a a
  to :: PF a a → a
```

In this definition $PF\ a$ is the functor which given a type will produce a new type $PF\ a\ a$. Note that every instance of Regular supplies its own functor which means that the *to/from* function must necesarily make it a type variable dependant on the input type. This is not possible in F# because type variables cannot accept type arguments.

4.3 Typeclasses and Generic Programming

Typeclasses are another Haskell specific feature essential for generic programming. They are the mechanism used in Haskell for function overloading. The special feature about typeclasses is the way they select function overloads. Consider the following portions of the *GInc* function:

```
instance GInc (K Int) where
  gInc (K i) = K (i + 1)
instance GInc Unit where
  gInc Unit = Unit
instance (GInc f, GInc g) ⇒ GInc (f ⊕ g) where
  gInc (Inl f) = Inl $ gInc f
  gInc (Inr g) = Inr $ gInc g
```

Consider what happens when *gInc* is invoked with a value of type $f \oplus g$. The function invokation makes a recursive call to an overload of *gInc* – but which? It is not possible to determine the precise overload until all type variables get instantiated. For instance, *gInc* can be called with a value of type $K\ Int \oplus Unit$ as well as a value of type $Unit \oplus Unit$ or even $GInc\ a \Rightarrow a \oplus K\ Int$. Each of these scenarios requires the compiler to select a different *gmap* overload. Haskell addresses the problem by performing type level computations when type variables get instantiated to select the correct overload.

In F#, method constraints could be used to achieve a similar result. For example, one could define the *GInc* funciton as an extension method of K, *Unit* and *Sum*:

```
type K⟨'t, 'x⟩ with
  member x.GInc () = x
```

```
type K⟨‘t, int⟩ with
  member x.GInc () = K (x.Elem + 1)
type Sum⟨‘t, ‘a, ‘b when
  (‘a : member GInc : unit → ‘a)
  and (‘b : member GInc : unit → ‘a)⟩ with
  member x.GInc () = match x.Elem with
    | Choice1Of2 v → Sum (Choice1Of2 v.GInc ())
    | Choice2Of2 v → Sum (Choice2Of2 v.GInc ())
```

However, type constraints in F# have the following limitations:

1. Extension methods are not checked by the compiler when solving type constraints
2. When extending a type, it must have the exact same signature as the original definition. The extension for $K\langle ‘t, int\rangle$ and *Sum* given above are not valid F# code.

These limitations (especially #2) highlight the additional type level computation power available in Haskell. To address them, F# would have to solve type constraints differently depending on how type variables are instantiated. Currently, type constraints in F# are solved the same way regardless on how the type variables of a type get instantiated.

4.4 Higher-Rank Polymorphism and Generic Programming

The RepLib [12] can be used to define generic functions for any Haskell 98 type. To achieve this it makes use of rank 2 types. The rank of a type is determined by the depth of nestings that the forall quantifier can appear. For example, RepLib defines the following rank-2 type:

```
data Con c a = ∀ l.Con (Emb l a) (MTup c l)
```

This type is rank-2 because the type $\forall$ quantifier of the l type is nested within the $\forall$ quantifier of the c and a types. To illustrate the usefulness of higher ranks in Haskell, consider a simpler rank-2 type:

```
data Rank2 = ∀ l.Rank2 l
```

With this type, one can define the following list:

```
[Rank2 5, Rank2 "String"] :: [Rank2]
```

Since every element of the list is of type *Rank2*. This is not very useful since any function that pattern matches on the *Rank2* constructor cannot perform any operation on the type it contains since it could be a value of any type. However, if type constraints are somehow imposed on the l, it would be possible to perform some operations. One possible way this could happen is through generalized algebraic data types (GADTs) [10]. Consider for example:

```
data V a where
  V :: Show a ⇒ a → V a
data Rank2 = ∀ l.Rank2 (V l)
```

In this case, the type V imposes the constraint that l will always be an instance of *Show*.

Back to RepLib, the type *Con* represents a type constructor. What RepLib does is that for every type a it represents, it defines a list of *Con* representing all the available constructors of that type. The argument l encodes the type of the arguments of the constructor. Since every constructor accepts arguments of different types, it must be a rank-2 type in order to have them all in the same list.

RepLib also defines a type *Emb*:

```
data Emb l a = Emb {
  to :: l → a,
  from :: a → Maybe l
}
```

This is the type used to pattern match constructors generically. For every type constructor, a value of type *Emb* is defined. This value contains two functions. The function *from* is used to generically pattern match this a value with a constructor. If the match is positive, it returns a value l inside a *Just*. As said before, l is used to encode the type of the arguments of the type-constructor. This allows the inclusion of the values given as parameters to the type constructor inside the *Just* constructor. Similarly, the *to* function takes a set of values with the type accepted by the type constructor and creates a new value using that constructor.

The RepLib library shows that rank polymorphism allows the programmer to specify how a generic function should behave depending on how the l argument is instantiated but at the same time, values that instantiate the l argument differently can be treated as if they were the same type.

4.5 Remarks

Typeclasses and GADTs give Haskell basic typelevel programming power. This allows the compiler to check that values constructed generically will be consistent with the type it represents. Withouth such typelevel programming capabilities, it is difficult to enforce correctness when constructing values generically since such correctness can only be checked at runtime with reflection.

5 Objectives

To explore the feasability of implementing a datatype generic programming in F#, the following objectives have been established:

General Objectives

- Implementing a library for datatype generic programming using Regular as a basis
- Compare the library to existing Haskell libraries
- Evaluate the library

Specific Objectives

- Define the types that will be used to define representations
- Create a mechanism to automatically derive representations
- Implement a mechanism to select method overloads using reflection
- Outline the shortcommings resulting from the lack of kind polymorphism
- Outline the shortcommings resulting from the lack of rank polymorphism
- Compare the library to Regular

```
[⟨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.

Part III
Strategy to Solve the Problem

6 Representations in F#

The core of datatype generic programming libraries is the representation type. As mentioned before, this library borrows its approach from Regular but has to be modified to cope with the limitations described in section 4.

All representations inherit from the class *Meta*. On a type level, the role of this class is to impose type constraints on type variables. Theese constraints are an alternative to the typeclass constraints used in Regular. For example, consider the following instance of the *GInc* defined above:

```
instance (GInc f, GInc g) ⇒
  GInc (f :*: g) where
    gInc f (x :*: y) = ...
```

Rather than abstracting over higher-kinded type arguments, this library abstracts over first-order type variables of kind $*$ and requires that they themselves are subtypes of the *Meta* class.

The concrete subtypes of *Meta* will be presented in the remainder of the section. Theese subtypes are analogous to the representation types already presented for Regular. 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 type constructors, analogous to $\oplus$ in Regular. Besides ‘*ty*, it takes two additional type arguments: ‘*a* and ‘*b*. It stores a single element of type $Choice\langle `a, `b\rangle$ which contains two type constructors: *Choice1Of2* and *Choice2Of2* which are used instead of *Inl* and *Inr*.

The second subclass of *Meta* is *Prod*, corresponding to the product of two types, analogous to $\otimes$ in Regular. Besides the ‘*ty* argument, the *Prod* type accepts two additional type arguments: ‘*a* and ‘*b* corresponding to the types of the two values of the product. It contains the properties *E1* and *E2* to access each of the elements of the product.

The third subclass of *Meta* is *K*, used to represent types that are not defined as ADTs, analogous to *K* in Regular. In addition to ‘*ty* it contains an extra argument ‘*a* which is the type of the value it contains. The variable ‘*a* has no type constraints since F# cannot statically constrain a type to not be an ADT. The value contained in *K* can be accessed by the property *Elem*.

The fourth subclass of *Meta* is *U*, used to represent empty type constructors, analogous to *U* in Regular. It takes no additional type arguments.

The fifth subclass of *Meta* is *Id*, used to represent recursion within a type, analogous to *I* in Regular. This type only contains a single value of type ‘*ty* which is the recursively occurring value.

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 the library can handle.

To show how representations look in F#, a similar list is defined in F#. This list has nested ADTs to highlight some of the differences between the representations of this library and Regular's.

```
type Shape = Square of int * in | Point
type SList = Cons of Shape * NatList
    | Nil


type SListRep =
    Sum⟨
        SList,
        Prod⟨SList, Sum⟨Shape, Prod⟨Shape, K⟨Shape, int⟩, K⟨Shape, int⟩, U⟨Shape⟩⟩⟩,
            Id⟨SList⟩⟩,
        U⟨Elems⟩⟩
```

It is important to notice that the first type argument ‘*ty* of any subclass of *FoldMeta* is instantiated to the type being represented. In the next sections it will be highlighted why this is important.

7 Automatic conversion between values and representations

Being able to convert values to and from representations automatically makes the library more convenient to use. Since this conversion is a single function, it can be implemented using reflection. The user of the library will not need to write reflective code to implement generic functions. This section describes how reflection can be used in F# to write the function.

Every object in .NET has a member function $\mathit{GetType} : \mathit{unit} \to \mathit{Type}$. This function returns a value of type *Type* containing all the metadata related to the type of the value. Many methods exist to inspect that metadata, most of them are available in the *Reflection* package of F#. Two important functions when dealing with ADTs are:

```
type FSharpType =
  static member IsUnion : Type → bool
  static member GetUnionCases : Type → List⟨UnionCaseInfo⟩
```

The *IsUnion* method checks at runtime whether or not values of the given type are defined as ADTs. The *GetUnionCases* method gives a list of all the constructors of an ADT. The *UnionCaseInfo* type contains information about each of the constructors and can be used to construct and pattern match values.

The remainder of this section describes the algorithm to convert values into representations. The code here intends to demostrate how the algorithm works and how reflection is used to implement it but the actual implementation is very different since this code omits lots of boilerplate code that arises from the use of reflection. It uses pseudo-code that has F# syntax but types are treated as first class values, it uses $\langle\rangle$ to distinguish types from values in the arguments of functions.

In this code, variables preceded by an apostrophe, such as $`x$, always refer to types, even when used as values. They are always of type $`x : \mathit{Type}$. This code also pattern matches types as if they were ordinary values. Reflection can mimick pattern matching on types because type objects can be checked for equality. However, polymorphic types cannot be directly compared to monomorphic types. For example, the types $\mathit{List}\langle`a\rangle$ and $\mathit{List}\langle\mathit{int}\rangle$ are not equal according to .NET. In order to match $\mathit{List}\langle\mathit{int}\rangle$ with the pattern $\mathit{List}\langle`a\rangle$, the method $\mathit{GetGenericTypeDefinition} : \mathit{unit} \to \mathit{Type}$ of the *Type* class is used to un-instantiate the type variables. The resulting type can be checked for equality.

Another simplification of the language is that it omits conversion from/to type variables (the types that appear inside $\langle\rangle$) to .NET values of type *Type*. This is straightforward to do. Suppose one has a generic function $\mathit{foo}\langle`t\rangle$, the *Type* object represented by the $`t$ variable can be obtained in the body of $\mathit{foo}\langle`t\rangle$ using the $\mathit{typeof}\langle`t\rangle : \mathit{Type}$ function.

The final simplification is that constructors for types that contain polymorphic type arguments are invoked by instantiating the polymorphic types with values of type *Type*. Suppose we wish to invoke the $\mathit{Sum}\langle`t, `a, `b\rangle$ constructor with the values $T, A, B : \mathit{Type}$ (which are runtime .NET values). First a *Type* object is constructed which instantiates $`t, `a, `b$ with T, A, B:

```
let sumTy′ = typeof ⟨Sum⟨obj, Meta, Meta⟩⟩
let sumTy = sumTy′.GetGenericTypeDefinition ().MakeGenericType ([| T; A; B |])
```

The *GetGenericTypeDefinition* member returns a *Type* object identical to the type it is invoked on but with all type variables un-instantiated. Then the *MakeGenericType* method instantiates all type variables of a type with other types which are given as arguments in an array. Nothe that the

MakeGenericType function is un-safe because it cannot check until runtime that the *Type* values passed as argument are compatible with the variables they are instantiating. With this new type, the constructor can be obtained using the $GetConstructor : Type\ [\,] \rightarrow ConstructorInfo$ and can then be called using the $Invoke : obj \rightarrow obj\ [\,] \rightarrow obj$ method. The first argument is the object on which a method is called (always *null* for constructors), the second argument is an array with the arguments passed to the constructor and it returns the object that gets constructed (in this case an object of type $Sum\langle T, A, B\rangle$). Note that this function is also unsafe because it dosen't check that the arguments given to the constructor are of the correct type and the resulting object from the *Invoke* function must be dynamically casted to the type that is expected to be produced. All these details are ignored in the pseudo-code and invoking constructors this way is simply done by using the notation $Sum\langle T, A, B\rangle(args)$ and assuming it returns the correct type.

Type representations are constructed in two stages. First the type of such representation is obtained by the *getTy* function. Then, given a value, a representation is constructed with the *to* function. The type of the representation is the type determined by the *getTy* function. Below are the signatures:

```
**let** $getTy : Type \rightarrow Type$
**let** $to : Type \rightarrow obj \rightarrow Meta$
```

Both of these functions only operate on ADTs. They are implemented in several stages. Below are the first two parts:

```
**let** $getTyUnion : \langle Type\rangle \rightarrow List\langle UnionCaseInfo\rangle \rightarrow Type$
    $getTyUnion\langle `t\rangle(uc :: [\,]) = getTyValue\langle `t\rangle\ \ uc$
    $getTyUnion\langle `t\rangle(uc :: ucs) = Sum\langle `t, getTyValue\langle `t\rangle\ \ uc, getTyUnion\langle `t\rangle\ \ ucs\rangle$


**let** $toUnion : \langle Type\rangle \rightarrow obj \rightarrow List\langle UnionCaseInfo\rangle \rightarrow Meta$
    $toUnion\langle Sum\langle `t, `a, `b\rangle\rangle(x)\ (uc :: ucs) =$
        **if** $uc.Matches\ x$ **then**
            $Sum\langle `t, `a, `b\rangle(toValue\langle `t\rangle x\ uc \rhd Choice1Of2)$
        **else**
            $Sum\langle `t, `a, `b\rangle(toUnion\langle `t, `b\rangle x\ ucs \rhd Choice2Of2)$
```

The *getTyUnion* function takes as arguments the type for which a representation will be computed and the information of the type constructors for that type encoded as a list of *UnionCaseInfo*. The function nests an application of the *Sum* type for every type constructor available in the argument type. For each of the type constructors, the function *getTyValue* is called. The *toUnion* function takes as arguments the type obtained by the *getTyUnion* function, the list of constructors and the value being converted to a representation. It tries to match the given value to the constructor. This is done using the *Matches* extension method of *UnionCaseInfo* type. This method is implemented using special methods generated by the F# compiler for every ADT which allow checking at runtime if a value was constructed using a particular type constructor. For each constructor that dosen't match, it applies a nesting of the *Sum* constructor and recursively calls itself with the next type argument of the *Sum* (the $\langle `b\rangle$) and the remaining constructors. When the match is positive, it provides the value and the matched constructor to the *toValue* function and packs the result in the corresponding *Sum*.

```
let $getTyValue : \langle Type \rangle \to UnionCaseInfo \to Type$
  $getTyValue\langle `t \rangle \ \ uc =$
    let $genTy\langle `ty \rangle =$
      if $FSharpType.IsUnion\langle `ty \rangle$ then $getTyUnion\langle `ty \rangle$
      else $K\langle `t, `ty \rangle$
    let $tys = uc.ArgumentsTypes$
    let $go\ (`ty :: tys) = Prod\langle `t, getTy\langle `ty \rangle, go\ tys \rangle$
      $go\ [`ty] = genTy\langle `ty \rangle$
      $go\ [\,] = U\langle `t \rangle$
    $go\ tys$

let $toValue : \langle Type \rangle \to `t \to UnionCaseInfo \to Meta$
  $toValue\langle Prod\langle `t, `a, `b \rangle \rangle (obj : `t)\ (uc : UnionCaseInfo) =$
    let $(args : List\langle obj \rangle) = uc.GetArguments\ obj$
    let $go\langle Prod\langle `t, `a, U\langle `t \rangle \rangle (x :: [\,]) = Prod\langle `t, `a, U\langle `t \rangle \rangle (to\langle `a \rangle \ \ x, U\langle `t \rangle ())$
      $go\langle Prod\langle `t, `a, `b \rangle (x :: xs) = Prod\langle `t, `a, `b \rangle (to\langle `a \rangle \ \ x, go\langle `b \rangle \ \ xs)$
    $go\langle Prod\langle `t, `a, `b \rangle \rangle \ \ args$
  $toValue\langle K\langle `t, `x \rangle \rangle (obj : `t)\ (uc : UnionCaseInfo) =$
    let $[v] = uc.GetArguments\ obj$
    $K\langle `t, `x \rangle (v)$
  $toValue\langle U\langle `t \rangle \rangle (obj : `t)\ (uc : UnionCaseInfo) = U\langle `t \rangle ()$
```

The *getTyValue* function handles the conversion of each of the type constructors to the type of the corresponding representation. It first extracts the type of each of the arguments of the type constructor. The code above uses the member function *ArgumentsTypes*. That function is not available in the reflection API but can be defined by querying the arguments accepted by the constructor method. Applications of the *Prod* constructor are nested for each argument accepted by the constructor. Each of the arguments is subsequently expanded into its representation which is done by calling the *getTyUnion* function for ADTs or using the *K* constructor for other types.

The *toValue* function looks involved but is also very simple. It is divided in three cases: *Prod*, *K* and *U*. The *K* case simply unpacks the only argument that is accepted by the constructor and packs the argument into the *K* constructor. The *U* case simply returns an instance of the $U\langle `t \rangle$ type. The *Prod* case extracts the value of all the arguments that were given to the type constructor. Again, the example uses the *GetArguments* member function which can be defined by invoking all the property accessors of the values accepted by the constructors. For each value, it applies the *Prod* constructor giving it as a first argument the representation of the value (obtained by calling the *to* function) and the recursive application of the function to serialize the remainder of the product. Finally we define the main functions:

```
let $to\langle Sum\langle `t, `a, `b \rangle \rangle obj = toUnion\langle Sum\langle `t, `a, `b \rangle \rangle obj\ FSharpType.GetUnionCases\langle `t \rangle$
let $to\langle Prod\langle `t, `a, `b \rangle \rangle obj = toValue\langle Prod\langle `t, `a, `b \rangle \rangle obj\ (head\ FSharpType.GetUnionCases\langle `t \rangle)$
let $to\langle K\langle `t, `x \rangle \rangle obj = toValue\langle K\langle `t, `x \rangle \rangle obj\ (head\ FSharpType.GetUnionCases\langle `t \rangle)$
let $to\langle U\langle `t \rangle \rangle obj = toValue\langle U\langle `t \rangle \rangle obj\ (head\ FSharpType.GetUnionCases\langle `t \rangle)$

let $getTy\langle `t \rangle =$
  if $FSharpType.IsUnion\langle `t \rangle$then
```

```
        getTyUnion⟨'t⟩
    else
        failwith "Not an ADT"
```

With these functions, conversion into a representation can be done by invoking the *getTy* function and passing the result to the *to* function along with the value which should be converted to a representation. Conversion from a representation into a value happens in a similar way but in the opposite direction. All this machinery is packed inside the *Generic* type which provides:

```
type Generic⟨'t⟩ =
    abstract To : 't → Meta
    abstract From : Meta → 't
```

```
[⟨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.

8 Defining generic functions as classes

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. The following section explains how the *FoldMeta* class is used to implement generic functions and provides some examples of implementations of common generic functions.

8.1 Overloading the *FoldMeta* class and *GMap* definition

Similar to Regular, generic functions will be defined by cases for each of the types that define representations. Since F# dosen't have typeclasses, each case will be defined by overriding methods of the abstract class called *FoldMeta*. The abstract definition of the *FoldMeta* is given in figure 2. The *FoldMeta* type is parametrized by the following type argumetns:

- *'t* which is the type being represented by the type representation
- *'in* which is the input type of the generic function
- *'out* which is the output type of the generic function

In addition to those arguments, the *Sum*, *Prod*, *K* and *U* variants of the method also include the type parameter *'ty*. Recall that all type representatios take as first type parameter the type being represented. In the case of *nested types* or types that contain within them other types, the parameter will vary in different sections of the representation. Therefore, it is necessary to quantify over all types, not only *'t*. Regular does not do this but it is necessary to define certain generic functions which will be covered later. The *K* override also contains the type parameter *'a* which denotes the primitive type contained by *K*.

This class can only handle generic functions that take a single argument. However, F# allows types to have the same name as long as they differ in the number of type parameters. This makes it possible to define variants of *FoldMeta* that take more arguments.

To illustrate how the library works. The generic function *GMap* will be used as an example. This function takes as an argument another function and applies the function on every occurence of the type of the function. The heading of the function is the following:

```
type GMap⟨‘t, ‘x⟩(f : ‘x → ‘x) =
  class
  inherit FoldMeta⟨
    ‘t,
    Meta⟩()
  end
```

This function uses the variant of *FoldMeta* that accepts no input arguments since the functional argument is moved to the constructor. It is easier to use class arguments if the argument dosen't change during recursive function calls. To perform the mapping, the function produces a new representation with updated values; hence the ‘*out* parameter is instantiated to *Meta*.

The first method that needs to be overriden is the *Sum* case:

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

The *Sum* constructor encodes the constructor that was used to create the value that was provided. The choice is encoded as nestings of the *Choice* type and the nesting is defined by using the *Choice1Of2* and *Choice2Of2* constructors. This override will recursively apply the *FoldMeta* function to both cases and pack the result back into a value with the same number of *Choice* nestings. The result must be casted to *Meta* in order to agree with the type of the method.

Next, the *Prod* case must be overriden:

```
override x.FoldMeta⟨‘ty⟩
  (v : Prod⟨‘ty, Meta, Meta⟩) =
    Prod⟨Meta, Meta⟩(
      x.FoldMeta (v.E1),
      x.FoldMeta (v.E2)
    ≺ Meta
```

The *Prod* type contains two properties, *E1* and *E2*, which correspond to the two representations from which a product is built. Again, the function only needs to be applied recursively to the inner representations of the product and then packed back.

To handle the *K* constructor, two methods are needed:

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

The first case handles the occurences of primitive values that have the same type as the input type of the argument function. It simply applies the function to the value and packs the result with the same constructor. The second case handles all other values. Since nothing can be done with them, they are returned as they are. Below is the definition for the *U* type which dosen't do anything special either.

Next, the *Id* case must be overriden:

```
override $x.FoldMeta$
  $(v : Id\langle `t\rangle) =$
    let $g = Generic\langle `t\rangle()$
    $Id\langle `t\rangle(x.FoldMeta\ ($
      $g.To\ c.Elem) \mathrel{|\!\!>} g.From)$
    $\prec Meta$
```

Since this library works with shallow representations, recursive values are not immediately converted to their representation. The *Id* constructor contains an instance of the type being represented. Since generic functions only work with representations, the value must first be converted to its representation, then *FoldMeta* can be recursively applied to the representation and finally the resulting representation is converted back to a value and packed inside the *Id* constructor.

Although the definition for *GMap* is complete, it is still incorrect. As it stands, it only allows primitive values to be mapped. Values that are expressible as a representation (ADTs) will not get mapped, just ignored. The reason is that such values get translated into their corresponding representation when the generic funciton gets applied. Here is were the first parameter of representation types becomes important. Three additional overloads are provided to map ADTs:

```
let $mapper\ (f : `x \rightarrow `x)\ (v : Meta) =$
  let $g = Generic\langle `x\rangle()$
  $v \mathrel{|\!\!>} g.From \mathrel{|\!\!>} f \mathrel{|\!\!>} g.To$
member $x.FoldMeta\ ($
  $u : U\langle `x\rangle, f : `x \rightarrow `x) = mapper\ f\ u$
member $x.FoldMeta\ ($
  $p : Prod\langle `x, Meta, Meta\rangle, f : `x \rightarrow `x) = mapper\ f\ p$
member $x.FoldMeta\ ($
  $s : Sum\langle `x, Meta, Meta\rangle, f : `x \rightarrow `x) = mapper\ f\ s$
```

Theese overloads match the type parameter of the representation type with the type of the first argument of the input function. When the match is positive, the function proceeds by calling the *mapper* helper function which converts the representation into a value, applies the function and converts the result back into a representation. Theese overloads no longer have the universally quantified '*ty* parameter since they work specifically for the type '*x* which gets instantiated at a class level rather than being instantiated when the method is invoked.

The definition is now correct and complete. If implemented with the library, it will generically map algerbaic data types. The following sections explain how the library correctly selects the methdos that are invoked in each case. Note that all recursive calls of the *FoldMeta* method invoke the overload with signature $FoldMeta : Meta \rightarrow `out$ for which no implementation was given. The implementation of the method is derived automatically using reflection and is explained in section 9.

8.2 Uniplate

One of the popular combinator based libraries is the Uniplate [7] library. All generic functions in this library are implemented on top of a single function called *uniplate*. Being able to implement the *uniplate* function with this library is a way to demostrate the expressiveness of the library.

The *uniplate* function takes as an argument a value and returns a tuple. The first element contains a list with all the recursive occurences of values of the same type as the input type within the input value. The second element of the tuple is a function that provided with a list of values, such as the list returned in the tuple, it constructs a new value. In Haskell, its signature is the following:

```
uniplate : Uniplate a ⇒ a → ([a], [a] → a)
```

The F# implementation 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])
```

The *uniplate* function will be implemented in terms of two functions. The first one is *Collect* which computes the list of all the recursive children of a type. The definition is the following:

```
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 definition is straightforward. The most important case is *Id*. Recall that *Id* indicates a recursive occurence of a value with the same type as the type that produced the current representation. Therefore that value is packed inside a list and returned as a result. The *K* and *U* simply return an empty list because they don't contain any relevant information for the *uniplate* function. The *Prod* case simply concatenates the results of the recursive invokation of the *Collect* function and the *Sum* case simply applys the function recursively and returns its result.

The second generic function requried for *uniplate* is *Instantiate*. This function takes as an argument the list of recursive occurences of values and produces a new value with the same type as the list's elements. The definition will be explained in parts. The header of the function is:

```
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
```

This function accepts in its constructor the list of values that will be used to instantiate the new value. It also defines a helper function *pop* which extracts a single element of that list. The *pop* function is used by the *Id* overload defined below:

```
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
```

This overload simply extracts the next element from the argument list and replaces the value provided by the representation with the value extracted from the list.

The cases 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⟨‘ty, Meta, Meta⟩(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
```

This definitions rely on the call-by-value semantics of F# since the *Prod* case assumes that *self.FoldMeta p.E1* will be evaluated before *self.FoldMeta p.E2*. In any case, both of these definitions recursively invoke the *Instantiate* function and return the result packed in the same fashion as the input.

Finally, the case for U and K are trivial since they don't modify their argument nor 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 wraps both of these functions into a single function which also handles value conversions:

```
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)
```

8.3 Generic Equality and Two Argument extension of *FoldMeta*

Generic equality is very common among generic functions but cannot be implemented with the *FoldMeta* class presented before. The reason it that generic equality requires two representation arguments. Fortunately, it is easy to extend the *FoldMeta* class to do recursion on more than one argument. This variant of *FoldMeta* will have the same name but is located in a different module. Below is an example of how to use the two argument version of *FoldMeta* to define generic equality:

```
type GEQ⟨‘t⟩() =
  inherit FoldMeta⟨‘t⟩()
  member x.FoldMeta⟨‘ty⟩(v1 : Sum⟨‘ty, Meta, Meta⟩, v2 : Sum⟨‘ty, Meta, Meta⟩) =
    match v1.Elem, v2.Elem with
     | Choice1Of2 v1‘, Choice1Of2 v2‘ → x.FoldMeta (v1‘, v2‘)
     | _ → false
  override x.FoldMeta⟨‘ty⟩(v1 : Sum⟨‘ty, Meta, Meta⟩, v2 : Meta) = false
  member x.FoldMeta⟨‘ty⟩(v1 : Prod⟨‘ty, Meta, Meta⟩, v2 : Prod⟨‘ty, Meta, Meta⟩) =
    x.FoldMeta (v1.E1, v2.E1) ∧ x.FoldMeta (v1.E2, v2.E2)
  override x.FoldMeta⟨ty⟩(v1 : Prod⟨‘ty, Meta, Meta⟩, v2 : Meta) = false
  member x.FoldMeta⟨‘ty⟩(v1 : U⟨‘ty⟩, v2 : U⟨‘ty⟩) = true
  override x.FoldMeta⟨‘ty⟩(v1 : U⟨‘ty⟩, v2 : Meta) = false
```

```
member   x.FoldMeta⟨'ty,'x⟩(v1 : K⟨'ty,'x⟩, v2 : K⟨'ty,'x⟩) =
   v1.Elem = v2.Elem
override x.FoldMeta⟨'ty,'x⟩(v1 : K⟨'ty,'x⟩, v2 : Meta) = false
member   x.FoldMeta (v1 : Id⟨'t⟩, v2 : Id⟨'t⟩) =
   let g = Generic⟨'t⟩()
   x.FoldMeta (v1.Elem |> g.From, v2.Elem |> g.From)
override x.FoldMeta (v1 : Id⟨'t⟩, v2 : Meta) = false
```

This definition is very straightforward to understand. When values of similar structure appear in the same place, they are compared for equality either with recursion or direct comparison like the *K* case. What is important about this definition is that *FoldMeta* variants that accept multiple representation arguments can be enforced to be complete by requiring an overload that instantiates all representation arguments excepting the first to *Meta*.

9 The *FoldMeta* class

The *FoldMeta* class is the interface to define generic functions. It has the purpose of ensuring that the definitions are complete and it also dispatches the correct methdod according to a custom set of type rules.

9.1 Enforcing complete definitions

Consider once again the *GInc* function that was previously defined using Regular. Assume that only the following cases were given:

```
instance GInc (K Int) where
  gInc (K i) = K (i + 1)
instance GInc U where
  gInc U = U
instance (GInc f, GInc g) ⇒ GInc (f ⊗ g) where
  gInc (f ⊗ g) = gInc f ⊗ gInc g
```

Consider these two types and their representations:

```
data T1 = T1 Int Int
data T1Rep = Prod (K Int) (K Int)
data T2 = T2 Int String
data T2Rep = Prod (K Int) (K String)
```

Values of type *T1* can be handled by the *GInc* function wheras values of type *T2* cannot since *GInc* lacks a case for *K String*. If one tried to apply *GInc* to a value of type *T2Rep*, the Haskell compiler would instantiate the variables and figure out that there is no *GInc* instance for *K String*. It was discussed in section 4.3 that F# cannot perform the necessary typelevel computations and that abstract members and member constraints cannot be used to dispatch the correct overloads. This means that the F# compiler has no way to check if a generic function can handle a particular representation.

The only option left is to require that every generic function handles every case. This is quite a drawback because generic functions in this library must be total for its universe – every value can be applied to every generic function as long as the value can be represented as an instance of *Meta*. As a result, the *FoldMeta* class requires an implementation for five methods which are able to handle all representations. More specialized overloads can be included and they will be used whenever the function's arguments are compatible with the method.

9.2 Overload Selection

The *GMap* function defined above has overlapping overloads – cases where several methods can be invoked for a particular value. This is a problem that many datatype generic libraries have. In the case of Haskell based libraries, the problem is usually solved by enabling the overlapping instances language extension.

In the case of F#, the problem must be approached differently. For starters, all overload selections must be statically resolved at compile time (as mentioned in section 4.3). For this reason, the F# language cannot support an extension similar to overlapping instances. However, this also restricts the library from allowing functions like *GMap* to be defined, which demand that a similar feature exists. To resolve the problem, a customized dispatch mechanism was created using reflection. This mechanism inspects, at runtime, the types of the arguments provided to the *FoldMeta* method and selects the correct overload based on selection rules. The rules are described in figure 3. This figure shows the type of the *FoldMeta* overload that will be selected based on the input type for the $\mathit{FoldMeta} : \mathit{Meta} \rightarrow \text{`}\mathit{out}$ overload. In this figure:

- $v \prec V$ denotes that the type T can be given to the value v.
- $T \in x$ dentoes that the object x has a *FoldMeta* overload with type T.
- The $\forall\text{`}ty$ notation is used to represent polymorphic types. In other words, the signatures $FoldMeta\ :\ \forall\text{`}ty\ .\ \text{`}ty \rightarrow X$ and $\mathit{FoldMeta}\langle\text{`}ty\rangle : \text{`}ty \rightarrow X$ are equivalent.

The figure describes a function defined by parts: the first column is used as a label, the second column is the result and the third column lists the conditions necessary to select the result on the second column on a particular input. The function takes as input the *FoldMeta* method call and returns the type that will be used to match a *FoldMeta* overload.

For example, consider a variant of *GMap* with the following overloads: i

```
type Dollars = Dollars of int
type GMap⟨'t,'a⟩(f : 'a → 'a) =
  member x.FoldMeta⟨'ty⟩ : K⟨'ty,'a⟩ → Meta
  member x.FoldMeta : K⟨Dollar, int⟩
  override x.FoldMeta⟨'ty,'x⟩ : K⟨'ty,'x⟩ → Meta
```

The the chosen overload is different depending on the first argument given to *FoldMeta*. Suppose *FoldMeta* is invoked with a value of type $K\langle \mathit{List}\langle \mathit{int}\rangle, \mathit{int}\rangle$. The applicable cases (available in figure 3) are: K1, K2, K3 and K4. Case K1 tries to find an overload that exactly matches the type of the input, in this case $K\langle \mathit{List}\langle \mathit{int}\rangle, \mathit{int}\rangle$, but no such overload exists. It then proceeds to the K2 case which matches any $\text{`}ty$ but fixes the type of the second variable of $K$ to some concrete type $V$. In the example above, if $\text{`}a$ is instantiated to *int*, then the overload with type $\forall\text{`}ty\ .\ K\langle\text{`}ty,\text{`}a\rangle$ is selected and the process finishes. If 'a is instantiated to any other type, there is no match. The process then proceeds to the K3 case. This case is identical to K2 but fixes the $\text{`}ty$ variable to a concrete type and matches any $\text{`}x$ in the second type variable of $K$. Needless to say, the K3 is not applicable to this example because no overload fixes $\text{`}ty$ to a concrete type and leaves $\text{`}x$ polymorphic. Finally, if none of the cases matches, the case K4 serves as a catch-all since its type can match any value of type $K\langle\text{`}ty,\text{`}x\rangle$. Similarly, when *FoldMeta* is invoked with a value of type $K\langle \mathit{Dollar}, \mathit{int}\rangle$ the case K1 finds an exact match since there is an overload that accepts values of type $K\langle \mathit{Dollar}, \mathit{int}\rangle$.

When many methods with compatible signature exist. Priority is first given to the closest match and then the position in the class hierarchy of the type that declared the candidate method. Although this mechanism is immitating the overlapping instances mechanism of the Haskell compiler, it gives the user a finer control to specify which method should be selected. In fact, this makes it trivial to extend or customize generic functions. For example, to define a function *GMapShallow* which does the same as *GMap* but does not traverse structures that occurr recursively, one can simply extend from *GMap* and override the *Id* case:

$$
x.FoldMeta(v) : `out = \left\{
\begin{array}{lcc}
\text{(S1)} & Sum\langle T, Meta, Meta\rangle \to `out & \begin{array}{c} v \prec Sum\langle T, Meta, Meta\rangle \\ Sum\langle T, Meta, Meta\rangle \to `out \in x \end{array} \\[2ex]
\text{(S2)} & \forall `ty\ .\ Sum\langle `ty, Meta, Meta\rangle \to `out & \exists `ty\ .\ v \prec Sum\langle `ty, Meta, Meta\rangle \\[2ex]
\text{(P1)} & Prod\langle T, Meta, Meta\rangle \to `out & \begin{array}{c} v \prec Prod\langle T, Meta, Meta\rangle \\ Prod\langle T, Meta, Meta\rangle \to `out \in x \end{array} \\[2ex]
\text{(P2)} & \forall `ty\ .\ Prod\langle `ty, Meta, Meta\rangle \to `out & \exists `ty\ .\ v \prec Prod\langle `ty, Meta, Meta\rangle \\[4ex]
\text{(K1)} & K\langle T, V\rangle \to `out & \begin{array}{c} v \prec K\langle T, V\rangle \\ K\langle T, V\rangle \to `out \in x \end{array} \\[2ex]
\text{(K2)} & \forall `ty\ .\ K\langle `ty, V\rangle \to `out & \begin{array}{c} \exists `ty\ .\ v \prec K\langle `ty, V\rangle \\ \forall `ty\ .\ K\langle `ty, V\rangle \to `out \in x \end{array} \\[2ex]
\text{(K3)} & \forall `x\ .\ K\langle T, `x\rangle \to `out & \begin{array}{c} \exists `x\ .\ v \prec K\langle T, `x\rangle \\ \forall `x\ .\ K\langle T, `x\rangle \to `out \in x \end{array} \\[2ex]
\text{(K4)} & \forall `ty, `x\ .\ K\langle `ty, `x\rangle \to `out & \begin{array}{c} \exists `ty, `x\ .\ v \prec K\langle `ty, `x\rangle \\ \forall `ty, `x\ .\ K\langle `ty, `x\rangle \to `out \in x \end{array} \\[2ex]
\text{(Id1)} & Id\langle T\rangle \to `out & \begin{array}{c} v \prec Id\langle T\rangle \\ Id\langle T\rangle \to `out \in x \end{array} \\[2ex]
\text{(U1)} & U\langle T\rangle \to `out & \begin{array}{c} v \prec U\langle T\rangle \\ U\langle T\rangle \to `out \in x \end{array} \\[2ex]
\text{(U2)} & \forall `ty\ .\ U\langle `ty\rangle \to `out & \exists `ty\ .\ v \prec U\langle `ty\rangle
\end{array}
\right.
$$

Figure 3: Selection criteria of the *FoldMeta* overload.

```
type $GMapShallow\langle `t, `x\rangle(f : `x \rightarrow `x) =$
  class
    inherit $GMap\langle `t, `x\rangle(f)$
    override $self.FoldMeta\ (v : Id\langle `t\rangle) = v$
  end
```

Here both functions can exist in the same namespace and context. In fact, a function could invoke both of them as if they were any two generic functions.

9.3 Limitations of the *FoldMeta* class

The most obvious limitation of the *FoldMeta* class is the number of arguments on which it can induct. For example, the generic equality function cannot be defined with *FoldMeta* as it stands since it must do recursion on two representations. To overcome the limitation, a variant of *FoldMeta* that performs induction on two of its arguments could be defined. The definition would look like:

```
[⟨AbstractClass⟩]
type $FoldMeta\langle `t, `out\rangle() =$
  abstract $FoldMeta : Meta * Meta \rightarrow `out$
  abstract $FoldMeta\langle `ty\rangle : Sum\langle `ty, Meta, Meta\rangle * Meta \rightarrow `out$
  abstract $FoldMeta\langle `ty\rangle : Prod\langle `ty, Meta, Meta\rangle * Meta \rightarrow `out$
  abstract $FoldMeta\langle `ty, `a\rangle : K\langle `ty, `a\rangle * Meta \rightarrow `out$
  abstract $FoldMeta : Id\langle `t\rangle * Meta \rightarrow `out$
  abstract $FoldMeta\langle `ty\rangle : U\langle `ty\rangle * Meta \rightarrow `out$
```

This definition ensures that all cases are covered when defining generic functions that accept two arguments. Additional overloads can be added to this class in order to pattern match specific cases. For example, when defining generic equality, one would like a method with type:

```
member  $FoldMeta\langle `ty\rangle : Sum\langle `ty, Meta, Meta\rangle * Sum\langle `ty, Meta, Meta\rangle \rightarrow `out$
```

which would recursively check each side of the sum for equality and return true if both sides are equal. This extension can be repeated to do recursion in any number of arguments. It is still limited by the fact that the library can only define a finite number of these extensions.

Another limitation of the *FoldMeta* class has to do with the type of values that can be returned by generic functions. Since generic functions are specified through the *FoldMeta* class, the return type of such functions is provided as a type argument to the class. This means that the return type of all cases must be the same. This is restrictive compared to other datatype generic programming libraries like Regular where the *Id* case might have a different return type as the *K* case. This is particularly important to ensure type safety on functions that construct values generically, such as *read*. The *FoldMeta* class cannot fully solve the problem without higher kinds. For example, to define *GMap* properly, one would like that the return type is the same as the input type. For the *K* would be:

```
abstract $FoldMeta\langle `ty, `a\rangle : K\langle `ty, `a\rangle \rightarrow K\langle `ty, `a\rangle$
```

Here, $`out$ gets instantiated to $K\langle `ty, `a\rangle$. Notice that both $`ty$ and $`a$ are universally quantified variables local to the *FoldMeta* definition, not the class. This means that in order for it to be possible to instantiate $`out$ to $K\langle `ty, `a\rangle$, $`out$ must be of kind $* \to * \to *$ since it must accept $`ty$ and $`a$ as arguments.

A possibility that could overcome some of the limitations is to extend the *FoldMeta* class with additional type arguments – one for each case. This would result in a new definition like:

```
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
  ⟩
```

This definition is still problematic since the return type of every overload is different. Recall that all overloads get dispatched by same method. This method has type $`m$, so it cannot return a value of type $`s$ or $`p$ since it results in a runtime error. To overcome this, one could add additional type constraints to ensure all return types are compatible with $`m$:

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

However, sub-type constraints cannot be enforced against type variables. This results in a compile time error since $`m$ is a type variable.

Part IV
Evaluation and Conclusions

10 Evaluation of the library in the F# language

One of the objectives is to asses the value generic programming can have for the F# programmer. The most important consideration is whether the library serves as a competitive approach to other means F# offers for implementing polytipic functions generically.

If a programmer needs to implement a polytipic function generically, he will typically have to use reflection. As mentioned in section 4, it has a lot of drawbeacks and will hardly become a tool for everyday use. The most important drawbacks from the generic programming point of view are:

- Error prone and type unsafe
- Requires a lot of boilerplate code
- Requires knowledge about the .NET platform
- Imperative programming style

The following section explores in greater detail these drawbacks and evaluates how our library addresses them.

On the positive side, this library provides a lightweight interface to define generic traversals. Generic traversals are defined simply by overriding the methdos of the *FoldMeta* class. Since those methods have well defined signatures and are implemented entirely in F#, the porgramer can benefit from all the type level features that the language offers. The main problem with generic traversals is that overlapping cases are not checked for type correctness until runtime. Nevertheless, it is easy to ensure that the type is correct since the function has the same signature as the abstract members but specialized to a type.

Since the interface of the *FoldMeta* class is much simpler that the interface of reflection and requires much less knowledge to be used, generic functions will probably have less bugs than functions implemented with reflection. On a more fundamentalist perspective, code that uses reflection looks highly imperative. It usually consists of invoking .NET routines in a specific order to obtain some data or invoke an operation. This is definitely not the way a functional first language should implement polytipic functions.

On the negative side, this library is much less expressive than reflection. It can only be used with ADTs – although it can handle other types embeded inside ADTs. Based on what was learned through this work, there is little hope that generic programming can work with classes sicne representations rely on objects being immutable. The reason is that a value and its representation are required to be an *embedding projection pair*. This means that $to \circ from \equiv id$ and $from \circ to \sqsubseteq id$ [2]. Classes may have mutable state and the state of an object cannot be recovered from the constructor that was used to create it. This means that a representation must contain information about every internal variable in a class rather than the constructor's arguments. Furthermore, the *Sum* constructor is probably useless since in the case of classes, it is not very relevant what constructor was used to create the value.

Classes are very important in F# code because other .NET languages do not support ADTs. It is often desirable that functions defined in F# also work on types defined in, for example, C#.

Since some objects are immutable, a special interface could be defined to specify how to translate and object from/to an ADT-like structure.

11 Evaluation of the library against Regular

Compared to Regular, this library has many shortcommings because of F#'s limited type system. Surprisingly, there are a couple of unexpected advantages comming from the use of reflection and the object oriented approach of this library. If F# supported kind-polymorphism, this library could be a competitive alternative to Regular.

The primary disadvantage occurs when values are constructed generically. Regular uses Haskell's type system to ensure that invalid representations will never be converted into values. The *to* and *from* functions use type families to give a unique type signature for every type that is an instance of *Regular*. It was pointed out in section 9.3 that this library can easily run into runtime errors since the compiler allows the *from* function of any *Generic* instance to be applied to any representation.

The lack of dependent types in F# leads to the possible scenario that a method dispatch might fail at runtime if a generic function is not total for all representations. Haskell addresses this issue by checking at compile time that a representation type is compatible with the generic function it is applied to. The only alternative to prevent method dispatch failures at runtime is to require that all generic functions are total for the universe. With this library one can still define partial functions that fail at runtime when applied to incompatible arguments but the advantage or Regular is that such runtime failure is converted to a compile time error.

On the performance side, this library must perform more work at runtime than Regular. Through cacheing, it is possible to achieve some performance gains but the first invokation will always require more work than with Regular. On the bright side, using the information available at runtime it is possible to dynamically generate code once which can be efficiently executed on further applications of a generic function. Optimization was not thoroghly studied in this thesis but generating the code dynamically might bring some of the benefits of *just in time* compilation to this library. It is left as future research how to optimize this implementation.

On the other hand, this library has some advantages over Regular. The most important one is being able to handle types that accept any number of type arguments. The reason is that Regular instances define a indexed type *Rep* that corresponds to representations. This type only allows representations that accept at most one type argument. In this library, all representations are of type *Meta*. This means they can take any number of arguments since they are hidden by the subtypeing relation. Although this library supports more type arguments, it is a consequence of it being less type safe.

Another nice advantage of this library is extensibility. As studied in section 9.2 it is easy to customize the behavior of generic functions by overriding generic methods. This is much harder to do in Haskell since only one instance per typeclasse is allowed. This advantage is also shared by the Scala implementation [9] of generic programming. The problem in Haskell relies in the fact that typeclasses are not well suited to define generic functions. Typeclasses are meant to express global properties of types but functions are local operations.

12 Remarks about F# and .NET

To develop this library, many features of F# where taken into consideration. This section talks about the limitations of some of F#'s features that made them un-suitable for generic programming.

12.1 Type Providers

Type providers are a mechanism in F# that can be used to generate types at compile time by executing .NET code. They use reflection to create instances of the *Type* class and those types are then included as if they were part of the program. Type providers support static parameters that can influence the types produced by the type provider. Type providers were initally designed to provide typed access to external data sources.

Type providers were considered as the first alternative to develop the library. The basic idea was that instead of having to provide several variants of the *FoldMeta* class accepting different number and kind of arguments, one could have a type provider that is able to generate many variants of the *FoldMeta* class to fulfill the requirements of many generic functions. This way, the programmer could specify as static parameters the number of parameters on which recursion should be done and the number of extra parameters accepted by the generic function.

Unfortunately, type providers are restricted on what types they are allowed to produce. Types that contain polymorphic type arguments cannot be generated with type providers. This means that no variant of the *FoldMeta* class is feasible with a type provider since it must at least accept the generic type as argument. This wouldn't be a problem if type providers could accept types as static arguments, but the only static arguments supported by type providers are strings, integers and booleans.

It is not unexpected that type providers are not enirely suitable for type level programming (they were designed with other objectives in mind) but the limitations show a lot of potential that F# could exploit by using reflection to generate types at compile time.

12.2 Add-Hoc Polymorphism

Ad-hoc polymorphism allows constraining polymorphic types to types that support a particular set of operations. This is the foundation on which Regular is built since generic functions are defined by extending the operations supported by representations. The same approach would have been the natural choice for this library, but F# deals with ad-hoc porlymorphism differently.

It is possible to add methods to types post-hoc (after the type has been defined), it can even be done in external modules. Since F# has memeber constraints, it should be possible to define generic functions as extension members of the representation types and use member constraints to enforce that the type variables corresponding to representation types support the generic function. However, member constraints in F# do not check if there exist extension members that satisfy the constraint. In F#, extension members are a convenient way to organize code but not a feature useful in the type system.

13 Conclusions and Future Work

It is well known that polytipic functions can lead to boilerplate code since they usually cannot be implemented generically with the constructs typically offered by functional languages. Since many of those functions are only dependent on the structure of types, it is possible to define algorithms that work on families of types. This has been achieved by methods such as datatype generic programming.

Generic programming has enjoyed lots of success in the Haskell programming language. It allows high levels of abstraction and uses the type system in an effective way to prevent ill-defined functions. Many methods even allow values to be constructed generically and ensure at compile time that the resulting representations are valid. The main drawback of generic programming is its relience on a powerful type system and immutability; making it hard to implement in other languages.

Even though F# is far away from having a type system that fully supports generic programming it runs on top of the .NET platform which provides a rich reflection api that can perform many of Haskell's type operations at runtime. Leveraging on reflection, it is possible to provide a safe interface that allows functions to be defined in a style similar to the generic programming approach of Regular.

This is evidenced by the library presented in this thesis along with some classic examples of generic functions. The interface provided by this library is easy to understand, provides some level of type safety and compared to reflection, which is typically used in F#, it has less room for errors. Functions are cleaner and more succint since the library eliminates the invokation of .NET internal routines. A more fundamentalist advantage over reflection is that it allows inductively defined functions.

Compared to Regular, it lacks many features due to F#'s simpler type system. The major flaw is that constructing values generically can easily lead to runtime errors since representations cannot be checked for correctness at compile time. Another shortcomming is that this library enforces that all generic functions are total for the universe. Regular allows partial functions since it can check at compile time that the function is not used on values for which the function is undefined. Finally, the library is less expressive since functions must be defined through the *FoldMeta* class and it is restricted on the arguments it can do induction with and values it can take as parameters.

On the other hand, this library confirms (as pointed out in [9]) that ideas from the object oriented world can benefit generic programming. In particular, method overriding is a powerful feature that allows the re-usage of existing generic functions to implement new generic functions by simply modifying individual induction cases. Alternatives to Haskell's typeclass approach should be considered by the Haskell community since typeclasses are quite rigid with the extensibility it provides.

This research shows that through reflection, one can immitate a lot of Haskell's type level computations. Currently, all reflection was carried out at run time in order to show what is possible with that framework. It would be interesting to research how much of theese computations could be performed post-compilation by implementing a tool that inspects the generated assemblies using reflection for correctness. This could potentially allow the programmer to define its own variations of the *FoldMeta* class while ensuring that the definition will not crash at runtime due to missing overloads. It would also be interesting to research possible runtime optimizaitions that can be done through reflection. Even though performance might never be on par to Regular (or a similar library) it could be possible that F#'s ability to generate code at runtime combined with the ideas from *just in time* compilation might lead to performance gains. This highlights a lot of the power that .NET's reflection framework has.

Bringing ideas of generic programming into everyday usage is a challenging work. F# is a nice playground because it allows programers (especially C# programmers) to switch into the language with minimal overhead. The language runs in Microsoft's .NET platform which has been deployed across many devices. This thesis shows that .NET's reflection API is capable of supporting many of the type level computations carried out by the Haskell compiler that are necessary for generic programming. The approach is far from complete compared to what is available in Haskell but there is room for improvement using the existing tools. Hopefully this thesis will inspire other researchers to investigate creative approaches to combine reflection and generic programming in a effective way.

Part V
References

References

[1] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, 2002.

[2] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and Theory. In *Generic Programming*, pages 1–56. Springer, 2003.

[3] Patrik Jansson and Johan 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.

[4] Ralf Lämmel and Simon 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.

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

[6] Microsoft. System.reflection namespace. `https://msdn.microsoft.com/en-us/library/system.reflection%28v=vs.110%29.aspx`.

[7] 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.

[8] Thomas Van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *International Conference on Functional Programming*, pages 13–24, 2008.

[9] 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.

[10] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In *In Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 09*, pages 341–352. ACM, 2009.

[11] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# 3.0.* Apress, November 2012.

[12] 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.

[13] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Lh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*, pages 233–244, 2009.