Generic Programming in F#

Datatype generic programming for .Net

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

Wouter Swierstra
Utrecht University
W.S.Swierstra@uu.nl

Abstract

The introduction of Datatype Generic programming (DGP) *revolutionized* functional programming by allowing numerous algorithms to be defined by induction over the structure of types while still providing type safety. Due to the advanced type system requirements for DGP, only a handful of functional languages can define generic functions making it inaccessible to most programmers. Ordinary languages provide reflection and duck typing as a mechanism to specify generic algorithms. These mechanisms are usually error prone and verbose. By combining ideas from DGP and implementing them through reflection, a type-safe interface to DGP has been built for the F# language. These generic algorithms can be accessed by any language running in the .Net platform.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

2. Background

2.1 The F# Language

The F# programming language is a functional language of the ML family. It focuses on being a productive language by levering on functional programming and at the same time easy to adopt by programmers of other .Net hosted languages. As a result, the lenguage has a much simpler type system than Haskell or Scala. Most of the development effort in the language has focused on features to work with data (like type-providers) and to be compatible with the .Net type system. Unlike Scala, F# performs no type errasure when compiled to the .Net platform.

There are several mechanism to define new types in F#: classes, records and algebraic data types. Classes correspond to the traditional object oriented paradigm and are allowed to inherit fields and functions from another type as long as the type is not sealed (which is a .Net attribute for types). Records and algebraic datatypes correspond to the functional approach of defining types. Records and ADTs are always sealed and can be pattern matched. All types in F# can define member functions (methods) and can implement any number of interfaces. Types can also have generic type arguments but they are required to be of kind $*$ (star).

2.2 The .Net platform

The .Net platform is a common runtime environment to allow the execution of a family of languages. It implements a very rich type system which includes support for generics. Many type operations that happen in F# (such as sub-typeing) are handled by the .Net platform. The sub-typeing relation will be denoted by $\tau_a :> \tau_b$ which means τ_a is a sub-type of τ_b and consequently a value of type τ_a can be automatically converted to a value of type τ_b.

Like most object oriented langagues, .Net sub-typeing mechanism that allows types to be automatically converted to types which are higher in the class hierarchy. The F# language uses the keyword `inherit` to denote that a type inherits from another type. A well known restriction of this mechanism is that sub-typeing rules cannot automatically be applied to generic type arguments. In other words $\tau_a :> \tau_b \nRightarrow T < \tau_a > :> T < \tau_b >$.

The .Net platform internally uses an abstract class `Type` to represent any of the types that are available. This class implements operations such as casting or instantiating the generic type arguments of a type. At runtime, any value can be queried for it's type and the program under execution can use this information and even produce new types while running. This collection of mechanisms is know nas reflection.

Since the `Type` class is not sealed. Languages can extend it with any information they want. This allows F# to include metadata specific for functional programming. Using this metadata, it is possible to determine at runtime what are the type constructors of an ADT and even pattern match the type on those constructors. It is also possible to invoke the constructor with the appropiate values to produce new values. It should be noted that doing so is not type safe since the compiler cannot check that the operations being preformed are safe and type errors will lead to runtime exceptions. Nevertheless, this mechanism is actively used in libraries such as FsPickler [**?**] which is a general purpose .Net serializer.

3. Type Representations in F#

The essence of DGP is to provide an abstraction that is able to treat values of different types and equivalent structure as equivalent while still providing type safety. Type representations are used to achieve that objective. The method used here is similar to the approach from Regular[?] but had to be adapted to cope with two of F#'s limitations:

- Generics of higher kind are not permitted in F# (nor .Net)
- Method calls must be resolved statically

All type representations are a sub-class of the `Meta` abstract class. It's main role is imposing type constraints on generics that are required to be a type-representation. Those constraints serve as an alternative to typeclass constraints that are used in Regular. For instance in Regular one might have:

```
instance (GenericClass a,GenericClass b) => GenericClass (a :*: b) where
  genericFunction x = ...
```

which in F# would translate to a class:

```
type GenericClass =
  member genericFunction<'a,'b when 'a :> Meta and 'b :> Meta>(x : Prod<'a,'b>) = ...
```

and this indicates that `'a` and `'b` must be a type representation. Later one will see that the constraints not need to appear in the signature of `genericFunction` because they are added to the `Prod` class itself (which is also shown later).

The first sub-class of `Meta` is `SumConstr`. This is used to represent the possible type constructors that an algebraic data type has. This type takes three type arguments: `t`,`a` and `b`. The first one indicates the type that this representation encodes (or `unit` when it is an intermediate representation). The `'a` argument corresponds to the type representation of values created by one of the type constructors. The `b` argument contains the representation of the remaining type constructors or serves the same role as `'a` to represent values created with the last type constructor. Both `'a` and `'b` have the constraint `'a, 'b :> Meta`. For instance suppose one has a type:

```
type Elems<'a> = Cons of 'a*Elems<'a>
               | Val of 'a
               | Nil
```

its representation would look like:

```
type ElemsRep<'a> = SumConstr<Elem<'a>,_,SumConstr<_,_>>
```

The second sub-class of `Meta` is `Prod`. This type is used to represent cases in which a type constructor accepts more than one argument. The `Prod` type accepts two type arguments: `'a` and `'b`. The first argument contains the type representation of one of the constructor's parameters. The second argument contains the representation of the remaining constructor's arguments or the type `U` which is used to denote emptyness. Both `'a` and `'b` have the constraint `'a, 'b :> Meta`. With this constructor it is possible to fill the blanks of `ElemsRep` as follows:

```
type ElemsRep<'a> = SumConstr<
  Elem<'a>,
  SumConstr<
    Prod<_,Prod<_,U>>,
    SumConstr<
      Prod<_,U>,
      U>>>
```

The third sub-class of `Meta` is `K`. This type is used to represent a type that is not an ADT. Such types cannot be generically manipulated with DGP, nevertheless it is possible to write algorithms that operate on ocurrences of a particular type(s) inside a ADT. The `K` constructor takes a single type argument `'a` which corresponds to the type of its content. Since F# cannot statically constrain a type to be or not to be an ADT, `'a` has no constraints. To continue with the example above, the type `Elem<int>` would be represented as:

```
type ElemsRep = SumConstr<
  Elem<int>,
  SumConstr<
    Prod<K<int>,Prod<_,U>>,
    SumConstr<
      Prod<K<int>,U>,
      U>>>
```

The fourth sub-class of `Meta` is `Id`. This type is used to represent recursion within a type. This is necessary otherwise a type representation would be infinite for recursive ADTs. This type takes a single type argument which is the which is the same type being represented. With this addition, `Elem<int>` is now represented as follows:

```
type ElemsRep = SumConstr<
  Elem<int>,
  SumConstr<
    Prod<K<int>,Prod<Id<Elem<int>>,U>>,
    SumConstr<
      Prod<K<int>,U>,
      U>>>
```

The last sub-class of `Meta` is `U`. This type is used to represent an empty argument in a type constructor. That is the reason the `Nil` constructor is represented as `U` and ocurrences of `Prod` will always have `U` as the second argument of the innermost `Prod`.

4. Generic Functions

The purpose of type representations is to provide an interface that the programmer can use to define generic functions. In other words is a language to define what the semantics of such functions are. The library will then be provided with a generic function definition and will apply it as appropiate to the values it is provided with.

To show how generic definitions look like, the generic function `gmap` will be defined. This function accepts as an argument a function of type $\tau \to \tau$ and applies the function to every value of type τ in a ADT. In Regular, a generic function is defined as a typeclass. In this implementation, they are defined as an ordinary .Net class:

```
type GMap<'t>(f : 't → 't) = class end
```

The class has a constructor that takes as argument the function that will be applied. The first step is dealing with the sum of type constructors. As explained in the previous section, they are represented by `SumConstr`:

```
member x.gmap<'x>(v : SumConstr<'t,Meta,Meta>) =
  match v with
  | L m → SumConstr<'x,Meta,Meta>(
            x.gmap m |> Choice1Of2)
  | R m → SumConstr<'x,Meta,Meta>(
            x.gmap m |> Choice2Of2)
```

Here the active patterns `L` and `R` are used to distinguish the two possible cases. Nevertheless, `gmap` is simply invoked recursively and the result is packed inside the same constructor. The next step is to deal with products. This is handled with the `Prod` constructor:

```
member x.gmap(v : Prod<Meta,Meta>) =
  Prod<Meta,Meta>(
    x.gmap(v.E1),
    x.gmap(v.E2))
```

The type `Prod` contains the properties `E1` and `E2` that access each of the elements of the product. Once again, `gmap` is invoked recursively on those values. Next is the case for the `K` constructor which contains values. Here is where the function gets applied:

```
member x.gmap(v : K<'t>) = K(f v.Elem)
```

The property `Elem` of the `K` constructor returns the value that is being represented by `K`. Note that this member only works for fundamental values that have a type that matches the argument function. Later it will be explained how other instances of `K` work.

The case for the `Id` constructor is a bit more involved because `Id` contains a property called `Elem` but the property contains a value, not a representation. In order to obtain the representation, the type `Generic<'t>` is provided. This type contains the members:

```
member x.To : 't → Meta
member x.From : Meta → 't
```

With that class it is now possible to extract the contents of `Id`, call the `gmap` function and convert the result back to the original type. This results in:

```
member x.gmap(v : Id<'t>) =
  let g = Generic<'t>()
  Id<'t>(x.Everywhere(
    g.To c.Elem) |> g.From)
```

There are still two pices missing in this generic function. First of all, the recursive calls are invoking `gmap` with an argument of type `Meta` and there is no overload that matches that type. Secondly, not all cases are covered. Addressing both of theese problems requires some boilerplate code and validations that cannot be checked by the compiler. To make matters simple, a type provider is used.

5. The Generic Type Provider

The F# language does not support generics of higher kind. This means that generic types in F# cannot be applied to other types. This feature is used by other DGP [**? ? ? ?**] to enforce type safety and allow the compiler to select

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...