

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 leveraging on functional programming and at the same time easy to adopt by programmers of other .Net hosted languages. As a result, the language 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 erasure 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-typing) are handled by the .Net platform. The sub-typing 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 languages, .Net sub-typing 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-typing rules cannot automatically be applied to generic type arguments. In other words $\tau_a :> \tau_b \not\Rightarrow T < \tau_a > :> T < \tau_b >$.

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 its type and the program under execution can use this information and even produce new types while running. This collection of mechanisms is known as 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 appropriate 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 performed 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CONF 'yy, Month d-d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...\$15.00.
<http://dx.doi.org/10.1145/nnnnnnnn.nnnnnnn>

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>,ConsRep,SumConstr<ValRep,NilRep>>` where `ConsRep`, `ValRep` and `NilRep` are the representations of each of the type constructors.

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

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