Teaching Generic Programming

Pieter Koopman Rinus Plasmeijer
Institute for Computing and Information Sciences (iCIS),
Radboud University Nijmegen, the Netherlands
pieter@cs.ru.nl rinus@cs.ru.nl

Draft

In this paper we explain how we teach generic programming to master students in computer science. Key concepts to transfer are the relation between plain types and their generic representation, kinds, and the function parameters corresponding to arrow kinds. We teach this successfully in three steps, we start with simple solutions, show why they fail and how a more sophisticated variant solves the problem. In this way the students understand why the generic system works as incorporated in Clean.

1 Introduction

We are responsible for the *advanced functional programming* in the master phase of the computer science curriculum at the Radboud University in Nijmegen and at ELTE in Budapest. We have chosen generic programming as one of the main topics of this course. Our goals are that students become familiar with using libraries that use generic programming, are able to construct generic algorithms themselves, and have a basic understanding of the rationale behind the generic system in Clean [5]. Insight in the principles of the generic structure facilitates the construction of generic algorithms.

In order to achieve these goals we use three versions of a generic system before we introduce the native generics of Clean [1]. For each version we show the possibilities and limitations. Those limitations are the reason to introduce a more sophisticated version of generic programming. For each version we offer programming exercises to illustrate the approach.

In our experience those steps are a successful preparation for understanding generic programming in Clean.

2 Preliminaries

The majority of the students in this master course in Nijmegen are students form our bachelor in computer science. Those students have participated in a 6 ec course in Algorithmics using C++, a 6 ec course in Object Oriented programming using Java, a 6 ec functional programming course using Clean, and a 3 ec course on the realization of data structures in Java. The master students are free to choose the advanced functional programming course, this implies that the students in the course are interested in functional programming. Usually the students have no active programming experience in functional programming for at least a year.

Apart from our own bachelor students there are usually various guests. Typically there are some students visiting Nijmegen in an exchange programme, as well as a few students form other departments in the faculty of science.

In Budapest we give only the genereric programming part of this course as part of the master curriculum in computer science.

3 First Approach: Class Based Generics

We start with the standard motivation for generic programming: avoiding tedious boilerplate code and make algorithms that work for any data type we will introduce in the future.

The first approach is a minimal version of generic programming. Ordinary algebraic types are used to construct a generic representation of data types.

```
:: UNIT = UNIT
:: PAIR a b = PAIR a b
:: EITHER a b = LEFT a | RIGHT b
```

The leading example in the lecture is equality as an ordinary class:

```
class == infix 4 t :: t t → Bool
```

Instances of this class for the basic generic types are:

```
instance == UNIT where (==) UNIT UNIT = True
instance == (EITHER a b) | == a & == b
where (==) (LEFT x)  (LEFT y)  = x == y
      (==) (RIGHT x) (RIGHT y) = x == y
      (==) _         _         = False
instance == (PAIR a b) | == a & == b
where (==) (PAIR x1 x2) (PAIR y1 y2) = x1 == y1 && x2 == y2
```

Instances of these classes for ordinary data types are made by manual transformation of the data types to their generic counterparts and comparing the generic representation of values. For instance, the ordinary data type `List a` had `GList a` as generic counterpart. The function `fromList` transforms any list to its generic representation. The instance of == for `List` just transforms the list to their generic version and compares those for equality.

```
:: List a  = Nil | Cons a (List a)
:: Glist a :== EITHER UNIT (PAIR a (List a))

fromList :: (List a) → GList a
fromList Nil         = LEFT UNIT
fromList (Cons a as) = RIGHT (PAIR a as)

instance == (List a) | == a
where (==) x y = fromList x == fromList y
```

Exercise

In the exercise we ask the students to implement a class for ordering in the classical way and the generic way.

```
:: Ordering = Smaller | Equal | Bigger
class (><) infix 4 a :: a a → Ordering
```

The lecture and the exercise illustrate the generic representation of data types and the transformation from (recursive) types and to a generic representation.

4 Second Approach: Cons Kind Generics

In the second lecture we extend the generics with constructor information in order to enable the printing of data structures. We add the generic type Con:

```
:: CON a = CON String a
```

Leading example is an overloaded operator to write values to a file:

```
class (<<<) infixl a :: *File a → *File
```

Instance of this operator are defined based on a manual transformation of data types to their generic representation. This example shows that it is sometimes necessary to have information about the constructors of the ordinary representation available in the generic representation. The names of constructors are missing in our first generic representation, this is fine for equality but a serious problem in any show function.

Kinds

As a more serious topic we introduce kinds in this lecture. As motivation example we introduce the type constructor class fmap:

```
class fmap t :: (a → b) (t a) → (t b)
```

The introduced generic approach fails here due to kind conflicts in the needed instance of fmap. The instance for UNIT does not need the function argument, but the instances for EITHER and PAIR need two of them. Only the instance for CON can be realized with the given function as argument.

We do not provide a solution for the kind problem with generics in this lecture. We explain in some detail the difference between type classes (kind $*$) and type constructor classes (kind $*\rightarrow*$), and the notion of kinds in general.

Exercise

In the exercise we ask the kinds of several types. Given a type like `List a`, as defined above, `List` has kind $*\rightarrow*$, while `List Int` has kind $*$.

Moreover we let the students implement instances for some recursive types for the classes show and parse.

```
class show  a :: a [String] → [String]
class parse a :: [String] → Maybe (a, [String])
```

The instances work via the generic route and all generic constructors are explicit. The constructors LEFT and RIGHT avoid backtracking in the parser. Using a list of strings instead of a file eliminates the need of a scanner.

The class show uses continuation to avoid excessive amount of appends. This is a useful programming technique shown on the fly.

This lecture and exercise illustrates the need for constructor information and make students familiar with the notion of kinds.

5 Third Approach: Generics for Higher Order Kinds

The students are now familiar with kinds and we show them how the problem with the generic map can be solved with a separate class map for each kind:

```
class map0 t ::                          t       → t       // t :: *
class map1 t :: (a → b)                 (t a)    → (t b)   // t :: * → *
class map2 t :: (a → b) (c → d) (t a c) → (t b d) // t :: * → * → *
```

Since we are looking for one general approach that can easily be applied by a compiler for every generic we make an equality in the same style:

```
class eq0 t ::                                    t       t       → Bool // t :: *
class eq1 t :: (a a → Bool)                      (t a)   (t a)    → Bool // t :: * → *
class eq2 t :: (a a → Bool) (b b → Bool) (t a b) (t a b) → Bool // t :: * → * → *
```

The instances of these classes for the generic types are:

```
instance eq0 UNIT   where eq0     _          _          = True
instance eq1 CON    where eq1 f   (CON _ x)  (CON _ y)  = f x y
instance eq2 PAIR   where eq2 f g (PAIR a b) (PAIR x y) = f a x && g b y
instance eq2 EITHER where eq2 f g (LEFT a)   (LEFT b)   = f a b
                          eq2 f g (RIGHT x)  (RIGHT y)  = g x y
                          eq2 _ _ _          _          = False
```

The instance of eq for a data type of kind $*\rightarrow*$ becomes just an application of eq1 parameterized with an appropriate comparison function for list elements. The instance of eq1 for List is based again on the generic representation of lists (EITHER (CON UNIT) (CON (PAIR a (List a)))):

```
instance eq (List a) | eq a
where eq l1 l2 = eq1 eq l1 l2

instance eq1 List
where eq1 eqa l1 l2 = eq2 (eq1 eq0) (eq1 (eq2 eqa (eq1 eqa))) (fromList l1) (fromList l2)
```

Although the first two arguments of eq2 look pretty complicated, their structure is completely determined by the generic representation of the type List. Hence, it is easy for a compiler to generate this. Note that we generate an instance of List in the generic representation and not an instance of List a (with kind $*$).

There is no separate exercise for this representation since it is combined with the next representation in a single lecture. The exercise of this lecture covers both representations.

6 Finally: Clean Generics

The step from the generics for higher order kinds to the native generics incorporated in Clean is very small.

1. The Clean compiler takes care of the necessary transformation from and to the generic representation.
2. In Clean we can use one generic for every kind (usually only the kinds $*$, $*\rightarrow*$ and $*\rightarrow*\rightarrow*$ occur). Base on the kind of the instance type the compiler knows how many additional arguments are needed.
3. The is more information for the CONS type than just the name. In fact there is a record that contains all information supplied in the corresponding algebraic type definition of this constructor.

4. There are some additional generic constructors like OBJECT (to hold type information) of objects, and FIELD to describe record fields in a generic way.

The examples shown are equality and map.

```
generic gEq  a    :: a a → Bool
generic gMap a b :: a   → b
```

Exercise

In the exercise we redo the show and parse example using three classes each: one for each kind. As a net step we ask an implementation without the generic tags. This typically requires backtracking in the instance of show2 for EITHER.

The instance of show for List becomes:

```
instance show1 List
where show1 sa l c = show2 (show1 show0) (show1 (show2 sa (show1 sa))) (fromList l) c
```

We stress that this rather complex instance is completely determines by the generic representation of List and hence can be derived easily by the compiler.

Next we ask the students to implement the show and parse as native generics in Clean:

```
generic show  a :: a [String] → [String]
generic parse a :: [String] → Maybe (a, [String])
```

7 Application of Generic Programming in the Course

During the rest of the course generic programming is used in most of the other topics treated.

The iTask system is built on a large part on generic programming [4]. In the treatment of the iTask system we focus on the construction of the domain specific language, dsl, for task oriented programming of work flow systems. It uses quite a number of generic functions under the hood. The user writing iTask programs notices this only by writing **derive class** iTask for the new data types she introduces.

The model-based test section of the course uses generics for the generation of test data, comparing test values, and printing them [2]. The generic generation of test values is treated in some detail. We show the basic generic generation algorithm as well as how to make tailor made instances of the algorithm to replace the generic behavior.

In the treatment of semantics [3] generic programming is used for printing syntax trees and by the test machinery to test semantic properties.

8 Conclusion

Generic programming is an important topic in our advanced functional programming course. Since it requires a new abstraction level it is a difficult topic to grasp for most students. We want to teach the students how to use given generic algorithms, how to define new generic functions themselves, and the rationale behind the generic system in Clean. Knowledge of alternative approaches to generic programming is out of scope for this course.

We use an iterative approach to explain generic programming. The first approach is based on ordinary type classes. It clearly illustrates the idea and the limitations of this approach. In two steps we extend this

to the generic architecture used in Clean. In our experience this route works well, it enables the students to use generic programming at the required level. We are curious to compare this way to teach generic programming with approaches used by others.

References

[1] Artem Alimarine & Rinus Plasmeijer (2002): *A generic programming extension for Clean*. In Thomas Arts & Markus Mohnen, editors: *Selected Papers of the 13th International Workshop on the Implementation of Functional Languages, IFL '01, Stockholm, Sweden*, *LNCS* 2312, Springer-Verlag, pp. 168–186.

[2] Pieter Koopman & Rinus Plasmeijer (2006): *Fully Automatic Testing with Functions as Specifications*, pp. 35–61. *LNCS* 4164, Springer-Verlag, Budapest, Hungary.

[3] Pieter Koopman, Rinus Plasmeijer & Peter Achten (2009): *An Effective Methodology for Defining Consistent Semantics of Complex Systems*, pp. 224–267. *LNCS* 6299, Springer-Verlag, Komarno, Slovakia.

[4] Rinus Plasmeijer, Peter Achten & Pieter Koopman (2008): *An introduction to iTasks: defining interactive work flows for the web*. In: *Revised Selected Lectures of the 2nd Central European Functional Programming School, CEFP '07*, *LNCS* 5161, Springer-Verlag, Cluj-Napoca, Romania, pp. 1–40.

[5] Rinus Plasmeijer & Marko van Eekelen (2002): *Clean language report (version 2.1)*. `http://clean.cs.ru.nl`.