A Generic Programming Extension for Clean

Artem Alimarine and Rinus Plasmeijer

Nijmegen Institute for Information and Computing Sciences,
1 Toernooiveld, 6525ED, Nijmegen, The Netherlands,
{alimarin,rinus}@cs.kun.nl

Abstract. Generic programming enables the programmer to define functions by induction on the structure of types. Defined once, such a generic function can be used to generate a specialized function for any user defined data type. Several ways to support generic programming in functional languages have been proposed, each with its own pros and cons. In this paper we describe a combination of two existing approaches, which has the advantages of both of them. In our approach overloaded functions with class variables of an arbitrary kind can be defined generically. A single generic definition defines a kind-indexed family of overloaded functions, one for each kind. For instance, the generic mapping function generates an overloaded mapping function for each kind.
Additionally, we propose a separate extension that allows to specify a customized instance of a generic function for a type in terms of the generated instance for that type.

1 Introduction

The standard library of a programming language normally defines functions like equality, pretty printers and parsers for standard data types. For each new user defined data type the programmer often has to provide similar functions for that data type. This is a monotone, error-prone and boring work that can take lots of time. Moreover, when such a data type is changed, the functions for that data type have to be changed as well. Generic programming enables the user to define a function once and specialize it to the data types he or she needs. The idea of generic programming is to define the functions by induction on the structure of types. This idea is based on the fact that a data type in many functional programming languages, including Clean, can be represented as a sum of products of types.

In this paper we present a design and implementation of a generic extension for Clean. Our work is mainly based on two other designs. The first is the generic extension for Glasgow Haskell, described by Hinze and Peyton Jones in [1]. The main idea is to automatically generate methods of a type class, e.g. equality. Thus, the user can define overloaded functions generically. The main limitation of this design is that it only supports type classes, whose class variables range over types of kind $\star$.

The second design described by Hinze in [2] is the one used in the Generic Haskell Prototype. In this approach generic functions have so-called kind-indexed

types. The approach works for any kind but the design does not provide a way to define overloaded functions generically.

The design presented here combines the benefits of the kind-indexed approach with those of overloading. Our contributions are:

- We propose a generic programming extension for Clean that allows for kind-indexed families of overloaded functions defined generically. A generic definition produces overloaded functions with class variables of any kind (though current implementation is limited to the second-order kind).
- We propose an additional extension, customized instances, that allows to specify a customized instance of a generic function for a type in terms of the generated instance for that type.

The paper is organized as follows. Section 2 gives an introduction to generic programming by means of examples. In Section 3 our approach is described. We show examples in Generic Clean and their translation to non-generic Clean. In section 4 we discuss the implementation in more detail. In section 5 we describe customized instances. Finally, we discuss related work and conclude.

2 Generic Programming

In this section we give a short and informal introduction to generic programming by example. First we define a couple of functions using type constructor classes. Then we discuss how these examples can be defined generically.

2.1 Type Constructor Classes

This subsection demonstrates how the equality function and the mapping function can be defined using overloading. These examples are the base for the rest of the paper. We will define the functions for the following data types:

```
:: List a       = Nil | Cons a (List a)
:: Tree a b     = Tip a | Bin b (Tree a b) (Tree a b)
```

The overloaded equality function for these data types can be defined in Clean as follows:

```
class eq t :: t t → Bool
instance eq (List a) | eq a where
    eq Nil Nil                      = True
    eq (Cons x xs) (Cons y ys)      = eq x y && eq xs ys
    eq x y                          = False
instance eq (Tree a b) | eq a & eq b where
    eq (Tip x) (Tip y)              = eq x y
    eq (Bin x lxs rxs) (Bin y lys rys) = eq x y && eq lxs lys && eq rxs rys
    eq x y                          = False
```

All these instances have one thing in common: they check that the data constructors of both compared objects are the same and that all the arguments of these constructors are equal. Note also that the context restrictions are needed for all the type arguments, because we call the equality functions for these types.

Another example of a type constructor class is the mapping function:

```
class fmap t :: (a → b) (t a) → (t b)
instance fmap List where
    fmap f Nil          = Nil
    fmap f (Cons x xs)  = Cons (f x) (fmap f xs)
```

The class variable of this class ranges over types of kind $\star \rightarrow \star$. In contrast, the class variable of equality ranges over types of kind $\star$. The tree type has kind $\star \rightarrow \star \rightarrow \star$. The mapping for a type of this kind takes two functions: one for each type argument.

```
class bimap t :: (a → b) (c → d) (t a c) → (t b d)
instance bimap Tree where
    bimap fx fy (Tip x)        = Tip (fx x)
    bimap fx fy (Bin y ls rs)  = Bin (fy y) (bimap fx fy ls) (bimap fx fy rs)
```

In general the mapping function for a type of arity n, takes n functions: one for each type argument. In particular, the mapping function for types of kind $\star$ is the identity function. This remark is important for section 3 where we define a mapping function for types of all kinds.

2.2 Generic Classes

In this subsection we show how to define the equality function generically, i.e. by induction on the structure of types. The user provides the generic definition of equality once. This definition can be used to produce the equality function for any specific data type. The approach described in this subsection assumes only generic definitions for classes, whose class variables range over types of kind $\star$. This is the approach described by Peyton Jones and Hinze in [1]. We present it here for didactic reasons. In the next section we will present our approach, based on Hinze's kind-indexed types [2], which does not have the limitation of kind $\star$.

The structure of a data type can be represented as a sum of products of types. For instance, a Clean data type

$$:: T\ a_1\ ...\ a_n\ = K_1\ t_{11}\ ...\ t_{1l_1} | \ ... \ | K_m\ t_{m1}\ ...\ t_{ml_m}$$

can be regarded as

$$T^{\circ}\ a_1\ ...\ a_n\ = (t_{11} \times \ldots \times t_{1l_1}) + \ldots + (t_{m1} \times \ldots \times t_{ml_m})$$

List and *Tree* from the previous section can be represented as

```
List° a       = 1 + a × (List a)
Tree° a b     = a + b × (Tree a b) × (Tree a b)
```

To encode such a representation in Clean we use the following types for binary sums and products.

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

N-ary sums and products can be represented as nested binary sums and products. The *UNIT* type is used to represent the product of zero elements, the *EITHER* type is a binary sum and the *PAIR* type is a binary product. With these types *List*$^\circ$ and *Tree*$^\circ$ can be represented as (in Clean a synonym type is introduced with :==)

```
:: List° a       :== EITHER UNIT (PAIR a (List a))
:: Tree° a b     :== EITHER a (PAIR b (PAIR (Tree a b) (Tree a b)))
```

Note that these types are not recursive. For instance, the right hand side of *List*$^\circ$ refers to the plain *List* rather than to *List*$^\circ$. So, the encoding affects only the "top-level" of a type definition. The recursive occurrences of *List* type are converted to List$^\circ$ "lazily". In this way it is easy to handle mutually recursive types (see [1]).

We need conversion functions to convert between a data type T and its generic representation T°. For example, the conversion functions for lists are

```
fromList                    :: (List a) → List° a
fromList Nil                = LEFT UNIT
fromList (Cons x xs)        = RIGHT (PAIR x xs)
toList                      :: (List° a) → List a
toList (LEFT UNIT)          = Nil
toList (RIGHT (PAIR x xs))  = Cons x xs
```

Now we are ready to define the equality generically. All the programmer has to do is to specify the instances for unit, sum, product and primitive types.

```
class eq t :: t t → Bool
instance eq Int where
    eq x y                      = eqInt x y
instance eq UNIT where
    eq UNIT UNIT                = True
instance eq (PAIR a b) | eq a & eq b where
    eq (PAIR x1 x2) (PAIR y1 y2) = eq x1 y1 && eq x2 y2
instance eq (EITHER a b) | eq a & eq b where
    eq (LEFT x) (LEFT y)        = eq x y
    eq (RIGHT x) (RIGHT y)      = eq x y
    eq x y                      = False
```

This definition is enough to produce the equality functions for almost all data types: an object of a data type can be (automatically) converted to the generic representation using the conversion functions and the generic representations

can be compared using the instances above. The integers are compared with the predefined function *eqInt*. We use integers as the only representative of primitive types. Other primitive types can be handled analogously. The *UNIT* type has only one inhabitant; the equality always return *True*. Pairs are compared component-wise. Binary sums are equal only when the constructors are equal and their arguments are equal. In general a data types may involve arrows. To handle such data types the user has to provide an instance on the arrow type ($\rightarrow$). Since equality cannot be sensibly defined for arrows, we have omitted the instance: comparing types containing arrows will result in a compile time overloading error.

These definitions can be used to produce instances for almost all data types. For instance, when the programmer wants the equality functions to be generated for lists and trees, (s)he specifies the following

instance *eq (List a)* **generic**
instance *eq (Tree a b)* **generic**

These definitions can be used to generate the following instances:

```
instance eq (List a) | eq a where
    eq x y = eq (fromList x) (fromList y)
instance eq (Tree a b) | eq a & eq b where
    eq x y = eq (fromTree x) (fromTree y)
```

So, we can implement the equality on arbitrary types using the equality on their generic representations. It is important to note that the way we convert the arguments and the results to and from the generic representation depends on the type of the generic function. The compiler generates these conversions automatically as described in section 4.4.

When we try to use the same approach to define *fmap* generically, we have a problem. The type language has to be extended for lambda abstractions on the type level. See [3] for details. Another problem is that we need to provide different mapping functions for different kinds: like *fmap* for kind $\star \rightarrow \star$, *bimap* for kind $\star \rightarrow \star \rightarrow \star$ and so on. Both of these problems are solved by the approach with kind-indexed types [2]. In our design, described in the following section, we use this approach in combination with type constructor classes.

3 Generics in Clean

In this section we show how generic functions can be defined and used in Clean. We use the mapping function as an example. To define the generic mapping function we write

```
generic map a1 a2                 :: a1 → a2
instance map Int where
    map x                         = x
instance map UNIT where
```

```
    map x                          = x
instance map PAIR where
    map mapx mapy (PAIR x y) = PAIR (mapx x) (mapy y)
instance map EITHER where
    map mapl mapr (LEFT x)   = LEFT (mapl x)
    map mapl mapr (RIGHT x)  = RIGHT (mapr x)
```

The **generic** definition introduces the type of the generic function. The **instance** definitions provide the mapping for the primitive types, *UNIT*, *PAIR* and *EITHER*.

The reader has probably noticed that the instances do not seem to "fit" together: they take a different number arguments. The function for integers takes no additional arguments, only the integer itself. Similarly, the function for *UNIT* takes only the *UNIT* argument; mapping for types of kind $\star$ is the identity function. The functions for *EITHER* and *PAIR* take two additional arguments; mapping for types of kind $\star \to \star \to \star$ needs two additional arguments: one for each type argument. The generic definition is actually a template that generates an infinite set of mapping classes, one class per kind. So, using the definition above we have defined

```
class map⋆ t          :: t → t
class map⋆→⋆ t        :: (a1 → a2) (t a1) → (t a2)
class map⋆→⋆→⋆ t      :: (a1 → a2) (b1 → b2) (t a1 b1) → (t a2 b2)
...
```

The class for kind $\star$ has the type of the identity function. The other two classes are renamings of *fmap* and *bimap* class from the previous section. The instances are bound to the classes according to the kind of the instance type.

```
instance map⋆ Int where
    map⋆ x                                = x
instance map⋆ UNIT where
    map⋆ x                                = x
instance map⋆→⋆→⋆ PAIR where
    map⋆→⋆→⋆ mapx mapy (PAIR x y) = PAIR (mapx x) (mapy y)
instance map⋆→⋆→⋆ EITHER where
    map⋆→⋆→⋆ mapl mapr (LEFT x)   = LEFT (mapl x)
    map⋆→⋆→⋆ mapl mapr (RIGHT x)  = RIGHT (mapr x)
```

The programmer does not have to write the kind indexes, they are assigned automatically by the compiler.

For convenience we introduce a type synonym for the type specified in the **generic** definition of mapping:

```
:: Map a1 a2 :== a1 → a2
```

The type of the generic mapping for a type of any kind can be computed using the following algorithm [2]:

$:: Map_\star\ t_1\ t_2 \quad :== Map\ t_1\ t_2$
$:: Map_{k \to l}\ t_1\ t_2 :== \forall a_1\ a_2.\ (Map_k\ a_1\ a_2) \to Map_l\ (t_1\ a_1)\ (t_2\ a_1)$

The mapping function for a type t of a kind k has type:

class $map_k\ t :: Map_k\ t\ t$

The type specified in a generic declaration, like *Map*, is called the polykinded type [2] of the generic function. We have to note that, though the type of map has two type arguments, the generated classes have only one class argument. It holds for all generic functions: the corresponding classes always have one class argument. It remains to be researched how to extend the approach for classes with more than one argument. In this example, we use type $Map_k\ t\ t$ with both arguments filled with the same variable t. It means that the consumed argument has the same top level structure as the produced result. We need two type variables to indicate that the structure does not have to be the same at the lower level. In the example of the reduce function at the end of this section we will give an idea about how to find the generic type of a function.

The programmer specifies which instances must be generated by the compiler. For *List* we write:

instance *map List* **generic**

The mapping for types of kind $\star \to \star$, like lists, can be used as usually, but the user now has to explicitly specify which map of the generated family of maps to apply. This is done by giving the kind between {| and |} as in

map$\{|\star \to \star|\}$ *inc (Cons 1 (Cons 2 (Cons 3 Nil)))*

Similarly, we can also get the mapping for type *Tree*, which is of kind $\star \to \star \to \star$.

instance *map Tree* **generic**

It can be used as in

map$\{|\star \to \star \to \star|\}$ *inc dec (Bin 1 (Tip 2) (Tip 3))*

In this example the values in the tips of the tree are incremented, the values in the branches of the tree are decremented. From now on for readability reasons we will write kind indexes as subscripts.

Let's go back to the equality example and see how to define generic equality in Clean:

```
generic eq t                                  :: t t → Bool
instance eq Int where
    eq x y                                    = eqInt x y
instance eq UNIT where
    eq x y                                    = True
instance eq PAIR where
    eq eqx eqy (PAIR x1 y1) (PAIR x2 y2) = eqx x1 x2 && eqy y1 y2
```

```
instance eq EITHER where
    eq eql eqr (LEFT x) (LEFT y)        = eql x y
    eq eql eqr (RIGHT x) (RIGHT y)      = eqr x y
    eq eql eqr x y                      = False
```

In this definition, like in the definition of map, the instances have additional arguments depending on the kind of the instance type. Again, the programmer specifies the instances to be generated, say:

```
instance eq List generic
instance eq Tree generic
```

This will result in two instances: $eq_{\star\to\star}$ *List* and $eq_{\star\to\star\to\star}$ *Tree*. The equality can be used as in

```
eq* eq* [1,2,3] [1,2,3] => True
eq*->* eq* eq* (Bin 1 (Tip 2) (Tip 3)) (Bin 1 (Tip 2) (Tip 3)) => True
eq* (λx y->eq* (length x) (length y)) [[1,2],[3,4]] [[1,1], [2,2]] => True
```

In the last line the two lists are equal if they are of the same length and the lengths of the element lists are equal.

One can see that this equality is more general than one defined in Section 2: the user can specify how to compare the elements of the structure. However, it is inconvenient to pass the "dictionaries" (such as $eq_\star$) manually every time. For this reason we generate additional instances that turn explicit dictionaries into implicit ones:

```
instance eq* (List a) | eq a where
    eq* x y = eq*->*->* eq* x y
instance eq* (Tree a b) | eq a & eq b where
    eq* x y = eq*->*->* eq* eq* x y
```

Such instances make it possible to call

```
eq* [1,2,3] [1,2,3]
eq* (Bin 1 (Tip 2) (Tip 3)) (Bin 1 (Tip 2) (Tip 3))
```

with the same effect as above.

The equality operator defined as a type class in the standard library of Clean can now be defined using the generic equality:

```
(==) infixr 5 :: t t -> Bool | eq* t
(==) x y = eq* x y
```

Consider an application of *map*

```
map*->* (λx -> 0) [[1,2], [3,4]]
```

What would it return: [0,0] or [[0,0], [0,0]]? The overloading will always choose the first. If the second is needed, the user has to write

$map_{\star\to\star}$ $(map_{\star\to\star}$ $(\lambda x \to 0))$ *[[1,2], [3,4]]*

As one more example we show the right reduce function, which is a generalization of *foldr* on lists. It takes a structure of type a and an "empty" value of type b and collapses the structure into another value of type b. Thus, the type is $a \to b \to b$, where a is the structure, i.e. a is a generic variable, and b is a parametrically polymorphic variable.

```
generic rreduce a :: a b → b
instance rreduce Int where
    rreduce x e = e
instance rreduce UNIT where
    rreduce x e = e
instance rreduce PAIR where
    rreduce redx redy (PAIR x y) e = redx x (redy y e)
instance rreduce EITHER where
    rreduce redl redr (LEFT x) e = redl x e
    rreduce redl redr (RIGHT x) e = redr x e
```

Reducing types of kind $\star$ just returns the "empty" value. The instance for pairs uses the result of reduction for the second element of the pair as the "empty" argument for reduction of the first element. To reduce the sum we just reduce the arguments.

The function is an example of a parametrically polymorphic function: here b is a non-generic type variable. We can define the standard *foldr* function that is defined on types of kind $\star \to \star$ using *rreduce*.

```
foldr :: (a b → b) b (t a) → b | rreduce_{⋆→⋆} t
foldr op e x = rreduce_{⋆→⋆} op x e
```

How do we come up with the type for generic reduce knowing the type of reduce for lists (*foldr*)? The type of a standard definition for *foldr* is:

foldr :: (a → b → b) b [a] → b

If it is generalized to any type of kind $\star \to \star$, it becomes

foldr :: (a → b → b) b (t a) → b

The type *(t a)* is the structure that we are collapsing. The first argument is the function that we apply to the elements of the structure, i.e. it is folding for type a of kind $\star$. So, we can choose the type $(a \to b \to b)$ as the generic type. With this generic type we get

```
class rreduce_⋆ a          ::  a b → b
class rreduce_{⋆→⋆} a      ::  (a_1 → b → b) (a a_1) b → b
class rreduce_{⋆→⋆→⋆} a    ::  (a_1 → b → b) (a_2 → b → b) (a a_1 a_2) b → b
```

The type for kind $\star \to \star$ is the same as the type of *foldr*, except that the last two arguments are flipped. This idea of finding out the generic type can be used for other functions that normally make sense for types of kind $\star \to \star$.

4 Implementation

In this section we describe how the generic definitions are translated to classes and instances of non-generic Clean. Suppose we need to specialize a generic function g to a data type T, i. e. generate an instance of g for T. A generic definition in general looks like

generic $g\ a_1\ ...\ a_r\ ::\ G\ a_1\ ...\ a_r\ p_1\ ...\ p_s$

Here G is the polykinded type of the generic function g, a_i are polykinded type variables and p_i are polymorphic type variables (i.e. the function is parametrically polymorphic with respect to them). We will denote $p_1\ \ldots\ p_s$ as $\boldsymbol{p}$.

To generate the instance for a data type T the following has to be specified by the user

instance $g\ T$ **generic**

A data type T has the following form

$:: T\ a_1\ ...\ a_n\ = K_1\ t_{11}\ ...\ t_{1l_1} | \ ... \ | K_m\ t_{m1}\ ...\ t_{ml_m}$

As an example in this section we will use the generic equality function defined in the previous section and its specialization to lists

:: Eq a :== a → a → Bool
generic *eq :: Eq a*
:: List a = Nil | Cons a (List a)
instance *eq List* **generic**

Here is a short summary of what is done to specialize a generic function g to a data type T. The following subsections give more details.

- Create the class g_k for the kind k of the data type T, if not already created. The instance on T becomes an instance of that class (Section 4.1).
- Build the generic representation T° for the type T. Also build the conversion functions between T and T° (Section 4.2)
- Build the specialization of g to the generic representation T°. We have all the ingredients needed to build the specialization because the type T° is defined using sums and products. The instances for sums and products are provided by the user as part of the generic definition. (Section 4.3).
- The generic function is now specialized to the generic representation T°, but we need to specialize it to the type T. We generate an adaptor that converts the function for T° into the function for T (Section 4.4).
- Build the specialization to the type T. It uses the specialization to T° and the adaptor. The instance g_k on T is the specialization of the generic function g to the type T. (Section 4.3).
- For convenience we additionally create shorthand instances for kind $\star$ (Section 4.5).

4.1 Kind-indexed Classes

The polykinded type of a generic function is used to compute the type of the function for a kind using the following algorithm [2]:

$$G_\star \quad t_1 \ ... \ t_r \ \boldsymbol{p} \ :== \ G \ t_1 \ ... \ t_r \ \boldsymbol{p}$$
$$G_{k_1 \to k_2} \ t_1 \ ... \ t_r \ \boldsymbol{p} \ :== \ \forall a_1 ... a_r . (G_{k_1} \ a_1 \ ... \ a_r \ \boldsymbol{p}) \to G_{k_2} \ (t_1 \ a_1) \ ... \ (t_r \ a_r) \ \boldsymbol{p}$$

From now on we will use the following shorthands:

$$G' \ t \ \boldsymbol{p} \ :== G \ t \ ... \ t \ \boldsymbol{p}$$
$$G'_k \ t \ \boldsymbol{p} \ :== G_k \ t \ ... \ t \ \boldsymbol{p}$$

where t in each right hand side occurs r times.

The generic extension translates a generic definition into a family of class definitions, one class per kind. The class has one class argument of kind k and one member. The type of the member is the polykinded type of the generic function specialized to kind k:

class $g_k \ t :: G'_k \ t \ \boldsymbol{p}$

Unlike [2], we use the polykinded types to type the class members rather than functions.

Each instance of a generic function is bound to one of the classes according to the kind of the instance type. For our example we have so far

class $eq_{\star \to \star} \ t :: (Eq \ a) \to Eq \ (t \ a)$
instance $eq_{\star \to \star}$ *List* **where** $eq_{\star \to \star} \ eqa = ...$

where the body of the instance is still to be generated.

4.2 Generic Type Representation

To specialize a generic function to a concrete data type one needs to build the generic representation of that type. This is rather straightforward. The algorithms of building the generic representation types and the conversion functions are described by Hinze in [4]. The conversion functions are packed into a record defined in the generic prelude:

$:: Iso \ a \ a^\circ = \{iso :: a \to a^\circ, \ osi :: a^\circ \to a\}$

Here we just give an example of the generic type representation and the isomorphism for the list type:

```
List° a :== EITHER UNIT (PAIR a (List a))
iso_List :: Iso (List a) (List° a)
iso_List = {iso=isoList,osi=osiList}
where isoList Nil                = LEFT UNIT
      isoList (Cons x xs)        = RIGHT (PAIR x xs)
      osiList (LEFT UNIT)        = Nil
      osiList (RIGHT (PAIR x xs)) = Cons x xs
```

4.3 Specialization

In this subsection we show how to specialize a generic function g to a data type T. It is done by first specializing it to the generic type representation T°. This specialization g_{T° is then used to build the specialization g_T to the data type T. The specialization to the generic T° is:

g_{T° *::* $G'_k\ T^\circ\ \boldsymbol{p}$
$g_{T^\circ}\ v_1\ \ldots\ v_n = \mathcal{S}(g, \{a_1 := v_1, \ldots, a_n := v_n\}, T^\circ)$

The following algorithm is used to generate the right hand side by induction on the structure of the generic representation:

$\mathcal{S}(g, \mathcal{E}, a)$	$= \mathcal{E}[a]$	*type variables*
$\mathcal{S}(g, \mathcal{E}, T)$	$= g_T$	*type constructors*
$\mathcal{S}(g, \mathcal{E}, t\ s)$	$= \mathcal{S}(g, \mathcal{E}, t)\ \mathcal{S}(g, \mathcal{E}, s)$	*type application*
$\mathcal{S}(g, \mathcal{E}, t \to s)$	$= g_{\to}\ \mathcal{S}(g, \mathcal{E}, t)\ \mathcal{S}(g, \mathcal{E}, s)$	*arrow type*

Type variables are interpreted as value variables bound in the environment $\mathcal{E}$, type constructors T as instances g_T of the generic function g on the data type T, type application as value application and arrow type as application of the instance of g for the arrow type. In [2] Hinze proves that the functions specialized in this way are well-typed. For the equality on *List*$^\circ$ the specialization is:

eq_{List° *:: (Eq a) → Eq (List*$^\circ$ *a)*
eq_{List° *eqa =* eq_{EITHER} eq_{UNIT} *(*eq_{PAIR} *eqa (*eq_{List} *eqa))*

The functions eq_{EITHER}, eq_{PAIR} and eq_{UNIT} are instances of the generic equality for the corresponding types. The function eq_{List} is the specialization to lists that we are generating.

The specialization to T is generated using the specialization to T°:

g_T *::* $G'_k\ T\ \boldsymbol{p}$
$g_T\ v_1\ \ldots\ v_n =$ *adaptor (*$g_{T^\circ}\ v_1\ \ldots\ v_n$*)*
where *adaptor :: (*$G'\ (T^\circ\ a_1\ \ldots\ a_n)\ \boldsymbol{p}) \to G'\ (T\ a_1\ \ldots\ a_n)\ \boldsymbol{p}$
adaptor = ...

The adaptor converts the function for T° into the function for T. The adaptors are generated using bidirectional mappings [1], described in the next subsection. The equality specialized to lists is

eq_{List} *:: (Eq a) → Eq (List a)*
eq_{List} *eqa = adaptor (*eq_{List° *eqa)*
where *adaptor :: (Eq (List*$^\circ$ *a)) → Eq (List a)*
adaptor = ...

The mutually recursive definitions of eq_{List} and eq_{List° show why we do not need type recursion in the generic type representation: the function converts lists to the generic representations as needed.

Now it is easy to fill in the instance for the type T. It is just the specialization to the type T.

instance $g_k\ T$ **where** $g_k = g_T$

The instance of the equality for lists is:

instance $eq_{\star\to\star}\ List$ **where** $eq_{\star\to\star} = eq_{\mathrm{List}}$

4.4 Adaptors

Adaptors are more complicated than one would expect. The reason is that generic function types and data types may contain arrows. Since the arrow type is contravariant in the argument position, we need bidirectional mapping functions to map it [1]. We define bidirectional mapping by induction on the structure of types as a special generic function predefined in the compiler:

generic *bmap a b :: Iso a b*

It is automatically specialized to all data types in the following way

instance *bmap T* **where**
bmap v_1 *...* v_n = {*iso=isoT, osi=osiT*}
where $isoT\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) = K_1\ x'_{11}\ \ldots\ x'_{1m_1}$
...
$isoT\ (K_m\ x_{m1}\ \ldots\ x_{ml_m}) = K_m\ x'_{m1}\ \ldots\ x'_{ml_m}$
$osiT\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) = K_1\ x''_{11}\ \ldots\ x''_{1m_1}$
...
$osiT\ (K_m\ x_{m1}\ \ldots\ x_{ml_m}) = K_m\ x''_{m1}\ \ldots\ x''_{ml_m}$

Here $x_{ij} :: t_{ij}$ is the jth argument of the data constructor K_i. New constructor arguments x'_{ij} and x''_{ij} are given by

$$x'_{ij} = (\mathcal{S}(bmap, \{a_1 := v_1, \ldots, a_n := v_n\}, t_{ij})).iso\ x_{ij}$$
$$x''_{ij} = (\mathcal{S}(bmap, \{a_1 := v_1, \ldots, a_n := v_n\}, t_{ij})).osi\ x_{ij}$$

The environment passed to $\mathcal{S}$ binds the type arguments of the data type T to the corresponding function arguments. For example, the instance for lists is

instance *bmap List* **where**
bmap v = {*iso=isoList, osi=osiList*}
where
isoList Nil = *Nil*
isoList (Cons x xs) = *Cons (v.iso x) (*$(bmap_{List}\ v)$*.iso xs)*
osiList Nil = *Nil*
osiList (Cons x xs) = *Cons (v.osi x) (*$(bmap_{List}\ v)$*.osi xs)*

The instance for the arrow is predefined as

instance *bmap* $(\to)$ **where**
bmap bmaparg bmapres = {*iso=isoArrow, osi=osiArrow*}
where *isoArrow f = bmapres.iso* $\cdot$ *f* $\cdot$ *bmaparg.osi*
osiArrow f = bmapres.osi $\cdot$ *f* $\cdot$ *bmaparg.iso*

This instance demonstrates the need for pairing the conversion functions together.

This generic function is used to build bidirectional mapping for a generic function type:

$$
\begin{aligned}
&bmap_g :: Iso_{kind(G)}\ G\ ...\ G\\
&bmap_g\ v_1\ \ldots\ v_r\ u_1\ \ldots\ u_s\\
&\quad = \mathcal{S}(bmap, \{a_1 := v_1, \ldots, a_r := v_r, p_1 := u_1, \ldots, p_s := u_s\}, G\ \boldsymbol{a}\ \boldsymbol{p})
\end{aligned}
$$

The function lifts the isomorphisms for the arguments to the isomorphism for the function type. In the function type the data type *Iso* is used as a polykinded type. It is instantiated to the type of the generic function G. The right hand side is defined by induction on the structure of type G. For the generic equality we have

$$
\begin{aligned}
&bmap_{\mathrm{eq}} :: (Iso\ a\ a^\circ) \to (Iso\ (Eq\ a)\ (Eq\ a^\circ))\\
&bmap_{\mathrm{eq}}\ v = bmap_{\to}\ v\ (bmap_{\to}\ v\ bmap_{\mathrm{Bool}})
\end{aligned}
$$

Bidirectional mapping for the primitive type *Bool* is the identity mapping, because it has kind $\star$.

Now we can generate the body of the adaptor:

$$adaptor = (bmap_g\ iso_T\ ...\ iso_T\ isoId\ ...\ isoId).osi$$

The a-arguments are filled in with the isomorphism for the data type T and the p-arguments with the identity isomorphism. In the current implementation ps are limited to kind $\star$, so we use the identity to map them. In our example of the equality on lists the adaptor is

$$adaptor = (bmap_{\mathrm{eq}}\ iso_{\mathrm{List}}).osi$$

4.5 Shorthand Instances for Kind $\star$

For each instance on a type of a kind other than $\star$ a shorthand instance for kind $\star$ is created. Consider the instance of a generic function g for a type $T\ a_1\ ...\ a_n$, $n \geq 1$. The kind k of the type T is $k = k_1 \to ... \to k_n \to \star$.

$$
\begin{aligned}
&\textbf{instance}\ g_\star\ (T\ a_1\ \ldots\ a_n) \mid g_{k_1}\ a_1\ \&\ \ldots\ \&\ g_{k_n}\ a_n\ \textbf{where}\\
&\quad g_\star = g_k\ g_{k_1}\ \cdots\ g_{k_n}
\end{aligned}
$$

For instance, for the equality on lists and trees we have

$$
\begin{aligned}
&\textbf{instance}\ eq_\star\ [a] \mid eq_\star\ a\ \textbf{where}\\
&\quad eq_\star\ x\ y = eq_{\star\to\star}\ eq_\star\ x\ y\\
&\textbf{instance}\ eq_\star\ Tree\ a\ b \mid eq_\star\ a\ \&\ eq_\star\ b\ \textbf{where}\\
&\quad eq_\star\ x\ y = eq_{\star\to\star\to\star}\ eq_\star\ eq_\star\ x\ y
\end{aligned}
$$

These instances make it is possible to call $eq_\star$ *[1,2,3] [1,2,3]* instead of $eq_{\star\to\star}\ eq_\star$ *[1,2,3] [1,2,3]*: they turn explicit arguments into dictionaries of the overloading system.

5 Customized Instances

Generic functions can be defined to perform a specific task on objects of a specific data type contained in any data structure. Such generic functions have the big advantage that they are invariant with respect to changes in the data structure.

Let's for example consider terms in a compiler.

```
:: Expr   = ELambda Var Expr
          | EVar Var
          | EApp Expr Expr
:: Var    = Var String
```

We can define a generic function to collect free variables in any data structure (e.g. parse tree):

```
generic fvs t                          :: t → [Var]
instance fvs UNIT where fvs x = []
instance fvs Int where fvs x  = []
instance fvs PAIR where
    fvs fvsx fvsy (PAIR x y)           = removeDup(fvsx x ++ fvsy y)
instance fvs EITHER where
    fvs fvsl fvsr (LEFT l)             = fvsl l
    fvs fvsl fvsr (RIGHT r)            = fvsr r
instance fvs Var where fvs x           = [x]
instance fvs Expr where
    fvs (ELambda var expr)             = removeMember var (fvs⋆ expr)
    fvs (EVar var)                     = fvs⋆ var
    fvs (EApp fun arg)                 = removeDup(fvs⋆ fun ++ fvs⋆ arg)
```

*UNIT*s and *Int*s do not contain variables, so the instances return empty lists. For pairs the variables are collected in both components; the concatenated list is returned after removing duplicates. For sums the variables are collected in the arguments. The instance on *Var* returns the variable as a singleton list. For lambda expressions we collect variables in the lambda body and filter out the lambda variable. For variables we call the instance on variables. For applications we collect the variables in the function and in the argument and return the concatenated list.

Now, if the structure containing expressions (e.g. a parse tree) changes, the same generic function can still be used to collect free variables in it. But if the expression type itself changes we have to modify the last instance of the function accordingly. Let's have a closer look at the last instance. Only the first alternative does something special - it filters out the bound variables. The other two alternatives just collect free variables in the arguments of the data constructors. Thus, except for lambda abstractions, the instance behaves as if it was generated by the generic extension. The generic extension provides a way to deal with this problem. The user can refer to the generic implementation of an instance that (s)he provides. In the example the instance on *Expr* can be written more compactly:

instance *fvs Expr* **where**
$$\begin{aligned} &\mathit{fvs}\ (\mathit{ELambda}\ \mathit{var}\ \mathit{expr}) = \mathit{removeMember}\ \mathit{var}\ (\mathit{fvs}_{\star}\ \mathit{expr}) \\ &\mathit{fvs}\ x \qquad\qquad\qquad\qquad\quad = \mathit{fvs}\{|\mathbf{generic}|\}\ x \end{aligned}$$

The name $\mathit{fvs}\{|\mathbf{generic}|\}$ is bound to the generic implementation of the instance in which it occurs. The code generated for the instance on *Expr* is:

$$\begin{aligned} &\mathit{fvs}^{g}_{\mathrm{Expr}}\ x \qquad\qquad\qquad\qquad\quad = (\mathit{bmap}_{\mathit{fvs}}\ \mathit{iso}_{\mathrm{Expr}}).\mathit{osi}\ (\mathit{fvs}_{\mathrm{Expr}^{\circ}}\ x) \\ &\mathit{fvs}_{\mathrm{Expr}}\ (\mathit{ELambda}\ \mathit{var}\ \mathit{expr}) = \mathit{removeMember}\ \mathit{var}\ (\mathit{fvs}_{\mathrm{Expr}}\ \mathit{expr}) \\ &\mathit{fvs}_{\mathrm{Expr}}\ x \qquad\qquad\qquad\qquad\quad = \mathit{fvs}^{g}_{\mathrm{Expr}}\ x \end{aligned}$$

Here $\mathit{fvs}^{g}_{\mathrm{Expr}}$ denotes the function generated for $\mathit{fvs}\{|\mathbf{generic}|\}$. The function for the generic representation $\mathit{fvs}_{\mathit{Expr}^{\circ}}$ is generated as usually.

6 Related Work

Generic Haskell is an extension for Haskell based on the approach of kind-indexed types, described in [2]. Despite pretty different notation, generic definitions in Generic Haskell and Clean are similar. The user provides the polykinded type and cases for sums, products, unit, arrow and primitive types. In Generic Haskell an overloaded function cannot be defined generically. It means that, for instance, equality operator (==) has to be defined manually. In Clean overloaded functions are supported. For instance, the equality operator in Clean can be defined in terms of the generic function *eq*:

$$\begin{aligned} &(==)\ \mathbf{infixr}\ 5 :: t\ t \to \mathit{Bool} \mid \mathit{eq}_{\star}\ t \\ &(==)\ x\ y = \mathit{eq}_{\star}\ x\ y \end{aligned}$$

Currently Generic Haskell does not support the module system. Clean supports the module system for generics in the same way as it does it for overloaded functions.

Glasgow Haskell supports generic programming as described in [1]. In GHC generic definitions are used to define default implementation of class members, giving systematic meaning to the **deriving** construct. Default methods can be derived for type classes whose class argument is of kind $\star$. That means that functions like mapping cannot be defined generically. In Clean a **generic** definition provides default implementation for methods of a kind-indexed family of classes. For instance, it possible in Clean to customize how elements of lists are compared:

$$\mathit{eq}_{\star\to\star}\ (\lambda x\ y \to \mathit{eq}_{\star}\ (\mathit{length}\ x)\ (\mathit{length}\ y))\ [[1,2],[3,4]]\ [[1,1],\ [2,2]] \Rightarrow \mathit{True}$$

This cannot be done in GHC, since the equality class is defined for types of kind $\star$. In Clean one generic definition is enough to generate functions for all (currently up to second-order) kinds. This is especially important for functions like mapping.

In [5] Chen and W. Appel describe an approach to implement specialization of generic functions using dictionary passing. Their work is a the intermediate

language level; our generic extension is a user level facility. Our implementation is based on type classes that are implemented using dictionaries. In SML/NJ the kind system of the language is extended, which we do not require.

PolyP [6] is a language extension for Haskell. It is a predecessor of Generic Haskell. PolyP supports a special **polytypic** construct, which is similar to our **generic** construct. In PolyP, to specify a generic function one needs to provide two additional cases: for type application and for type recursion. PolyP generic functions are restricted to work on regular types. A significant advantage of PolyP is that recursion schemes like catamorphisms (folds) can be defined. It remains to be seen how to support such recursion schemes in Clean.

In [8] Lämmel, Visser and Kort propose a way to deal with generalized folds on large systems of mutually recursive data types. The idea is that a fold algebra is separated in a basic fold algebra and updates to the basic algebra. The basic algebras model generic behavior, whereas updates to the basic algebras model specific behavior. Existing generic programming extensions, including ours, allow for type indexed functions, whereas their approach needs type-indexed algebras. Our customized instances (see section 5) provide a simple solution for dealing with type-preserving (map-like) algebras (see [8]). To support type-unifying (fold-like) algebras we need more flexible encoding of the generic type representation.

7 Conclusions and Future Work

In this paper we have presented a generic extension for Clean that allows to define overloaded functions with class variables of any kind generically. A generic definition generates a family of kind-indexed type (constructor) classes, where the class variable of each class ranges over types of the corresponding kind. For instance, a generic definition of map defines overloaded mapping functions for functors, bifunctors etc. Our contribution is in extending the approach of kind-indexed types [2] with overloading.

Additionally, we have presented an extension that allows for customization of generated instances. A custom instance on a type may refer to the generated function for that type. With this feature a combination of generic and specific behavior can be expressed.

Currently our prototype lacks optimization of the generated code. The overhead introduced by the generic representation, the conversion functions and the adaptors is in most cases unacceptable. But we are convinced that a partial evaluator can optimize out this overhead and yield code comparable with hand-written one. Our group is working on such an optimizer.

Generic Clean currently cannot generate instances on array types and types of a kind higher then order 2. Class contexts in polykinded types are not yet supported. To support pretty printers and parsers the data constructor information has to be stored in the generic type representation. The current prototype has a rudimentary support for uniqueness typing. Uniqueness typing in polykinded types must be formalized and implemented in the compiler. As noted in section

6 our design does not support recursion schemes like catamorphisms. We plan to add the support in the future.

Acknowledgements. We are grateful to Sjaak Smetsers for fruitful discussions and comments on this paper. For helpful comments we are also grateful to Peter Achten, Marko van Eekelen and three anonymous referees. We would like to thank Ralf Hinze for the discussion of customized instances.

References

1. Ralf Hinze and Simon Peyton Jones. *Derivable Type Classes.* In Graham Hutton, editor, Proceedings of the Fourth Haskell Workshop, Montreal, Canada, September 17, 2000
2. Ralf Hinze. *Polytypic values possess polykinded types.* In Roland Backhouse, J.N. Oliveira, editors, Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), Ponte de Lima, Portugal, July 3-5, 2000.
3. Ralf Hinze. *A New Approach to Generic Functional Programming.* In Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21, 2000.
4. Ralf Hinze. *A Generic Programming Extension for Haskell* In Erik Meijer, editor, Proceedings of the Third Haskell Workshop, Paris, France, September 1999. The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.
5. Juan Chen and Andrew W. Appel. *Dictionary Passing for Polytypic Polymorphism* Princeton University Computer Science TR-635-01, March 2001.
6. P. Jansson and J. Jeuring, *PolyP - a polytypic programming language extension*, POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press 470–482, 1997.
7. M. J. Plasmeijer, M.C.J.D. van Eekelen *Language Report Concurrent Clean. Version 1.3.* Technical Report CSI R9816, Faculty of mathematics and Informatics, Catholic University of Nijmegen, June 1998. Also available at www.cs.kun.nl/~clean/Manuals/manuals.html
8. Ralf Lämmel, Joost Visser, and Jan Kort. *Dealing with large bananas.* In Johan Jeuring, editor, Workshop on Generic Programming, Ponte de Lima, July 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.