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. The approach with kind-indexed types makes it possible to define generic functions indexed by types of different kinds. Another approach allows to define default implementation for instances of type classes in a generic way. In this paper we describe a combination of these two approaches, which has advantages of both of them. The key idea of our approach is that a generic function generates a kind-indexed system of type classes, one class per kind. The class variable of such a class ranges over types of the corresponding kind. For instance, an overloaded equality operator can be defined as a specific case of a generic equality function for kind $\star$.
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 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 [3]. 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 does not support type classes, whose class variables range over a type of a kind higher than $\star$.

The second design is the one used by Generic Haskell Prototype. In this approach generic functions have so-called kind-indexed types. Hinze describes it in [4]. The approach works for any kind, one generic definition is enough to generate functions for types of all kinds. 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 first-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.

See sections 7 and 8 for more detailed discussion of related work and contributions.

The rest of 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. Integration with the module system is discussed in section 6. 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:

$$
\begin{array}{ll}
{::}\ \mathit{List}\ a & = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
{::}\ \mathit{Tree}\ a\ b & = \mathit{Tip}\ a \mid \mathit{Bin}\ b\ (\mathit{Tree}\ a\ b)\ (\mathit{Tree}\ a\ b) \\
{::}\ \mathit{Rose}\ a & = \mathit{Rose}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{eq}\ t :: t\ t \rightarrow \mathit{Bool} \\
\textbf{instance}\ \mathit{eq}\ (\mathit{List}\ a) \mid \mathit{eq}\ a\ \textbf{where} \\
\qquad \mathit{eq}\ \mathit{Nil}\ \mathit{Nil} \qquad\qquad\qquad\qquad = \mathit{True}
\end{array}
$$

$$
\begin{array}{ll}
\quad \mathit{eq}\ (\mathit{Cons}\ x\ \mathit{xs})\ (\mathit{Cons}\ y\ \mathit{ys}) & = \mathit{eq}\ x\ y\ \&\&\ \mathit{eq}\ \mathit{xs}\ \mathit{ys} \\
\quad \mathit{eq}\ x\ y & = \mathit{False} \\
\multicolumn{2}{l}{\textbf{instance}\ \mathit{eq}\ (\mathit{Tree}\ a\ b) \mid \mathit{eq}\ a\ \&\ \mathit{eq}\ b\ \textbf{where}} \\
\quad \mathit{eq}\ (\mathit{Tip}\ x)\ (\mathit{Tip}\ y) & = \mathit{eq}\ x\ y \\
\quad \mathit{eq}\ (\mathit{Bin}\ x\ \mathit{lxs}\ \mathit{rxs})\ (\mathit{Bin}\ y\ \mathit{lys}\ \mathit{rys}) & = \mathit{eq}\ x\ y\ \&\&\ \mathit{eq}\ \mathit{lxs}\ \mathit{lys}\ \&\&\ \mathit{eq}\ \mathit{rxs}\ \mathit{rys} \\
\quad \mathit{eq}\ x\ y & = \mathit{False} \\
\multicolumn{2}{l}{\textbf{instance}\ \mathit{eq}\ (\mathit{Rose}\ a) \mid \mathit{eq}\ a\ \textbf{where}} \\
\quad \mathit{eq}\ (\mathit{Rose}\ x\ \mathit{xs})\ (\mathit{Rose}\ y\ \mathit{ys}) & = \mathit{eq}\ x\ y\ \&\&\ \mathit{eq}\ \mathit{xs}\ \mathit{ys}
\end{array}
$$

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:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class}\ \mathit{fmap}\ t :: (a \to b)\ (t\ a) \to (t\ b)} \\
\multicolumn{2}{l}{\textbf{instance}\ \mathit{fmap}\ \mathit{List}\ \textbf{where}} \\
\quad \mathit{fmap}\ f\ \mathit{Nil} & = \mathit{Nil} \\
\quad \mathit{fmap}\ f\ (\mathit{Cons}\ x\ \mathit{xs}) & = \mathit{Cons}\ (f\ x)\ (\mathit{fmap}\ f\ \mathit{xs}) \\
\multicolumn{2}{l}{\textbf{instance}\ \mathit{fmap}\ \mathit{Rose}\ \textbf{where}} \\
\quad \mathit{fmap}\ f\ (\mathit{Rose}\ x\ \mathit{xs}) & = \mathit{Rose}\ (f\ \mathit{xs})\ (\mathit{fmap}\ f\ \mathit{xs})
\end{array}
$$

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{class}\ \mathit{bimap}\ t :: (a \to b)\ (c \to d)\ (t\ a\ c) \to (t\ b\ d)} \\
\multicolumn{2}{l}{\textbf{instance}\ \mathit{fmap}\ \mathit{Tree}\ \textbf{where}} \\
\quad \mathit{bimap}\ \mathit{fx}\ \mathit{fy}\ (\mathit{Tip}\ x) & = \mathit{Tip}\ (\mathit{fx}\ x) \\
\quad \mathit{bimap}\ \mathit{fx}\ \mathit{fy}\ (\mathit{Bin}\ y\ \mathit{ls}\ \mathit{rs}) & = \mathit{Bin}\ (\mathit{fy}\ y)\ (\mathit{bimap}\ \mathit{fx}\ \mathit{fy}\ \mathit{ls})\ (\mathit{bimap}\ \mathit{fx}\ \mathit{fy}\ \mathit{rs})
\end{array}
$$

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 [3]. We present it here for didactic reasons. In the next section we will present our approach, based on Hinze's kind-indexed types [4], 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

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

For instance, *List*, *Tree* and *Rose* from the previous section can be represented as

$$\begin{array}{ll}
\mathit{List}^\circ\ a & = 1 + a \times (\mathit{List}\ a) \\
\mathit{Tree}^\circ\ a\ b & = a + b \times (\mathit{Tree}\ a\ b) \times (\mathit{Tree}\ a\ b) \\
\mathit{Rose}^\circ\ a & = a \times \mathit{List}\ (\mathit{Rose}\ a)
\end{array}$$

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 $List^\circ$, $Tree^\circ$ and $Rose^\circ$ 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)))
:: Rose° a       :== PAIR a (List (Rose a))
```

Note that these types are not recursive. For instance, the right hand side of $List^\circ$ refers to *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° "lazily". This way it is easy to handle mutually recursive types (see [3]).

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 x y = *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 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 equality functions to be generated for lists, trees and rose trees, (s)he specifies the following

instance *eq (List a)* **generic**
instance *eq (Tree a b)* **generic**
instance *eq (Rose a)* **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)*
instance *eq (Rose a)* | *eq a* **where**
    eq x y = *eq (fromRose x) (fromRose y)*

We have implemented the equality on 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. To illustrate it, we define a pair of functions: *encode* and *decode*. The first encodes a value as a list of bits and the second decodes a value from a list of bits. The functions can be regarded as a very simple “pretty” printer and parser.

class *encode t* :: *t* $\rightarrow$ *[Bit]*
instance *encode Int* **where**

```
    encode x              = encodeInt x
instance encode UNIT where
    encode x              = []
instance encode (PAIR a b) | encode a & encode b where
    encode (PAIR x y) = encode x ++ encode y
instance encode (EITHER a b) | encode a & encode b where
    encode (LEFT x)    = [0: encode x]
    encode (RIGHT x) = [1: encode x]
```

To encode *Int* we use a function defined elsewhere. The *UNIT* type has one constructor and, thus, can be encoded with no bits. *PAIR* has one constructor: no bits are needed to encode it. The encodings for the arguments of pairs are concatenated. *EITHER* has two constructors, one bit is enough to encode them. The *decode* function decodes an object encoded by the *encode* function:

```
class decode t :: [Bit] → (t, [Bit])
instance decode Int where
    decode x     = decodeInt x
instance encode UNIT where
    decode x     = (UNIT, x)
instance decode (PAIR a b) | decode a & decode b where
    decode x     = let  (l, x1) = decode x
                        (r, x2) = decode x1
                   in   (PAIR l r, x2)
instance encode (EITHER a b) | decode a & decode b where
    decode [0:x] = let(l, x) = decode x in(LEFT l, x)
    decode [1:x] = let(r, x) = decode x in(RIGHT r, x)
    decode []    = abort "cannot decode"
```

The would-be-generated instances of *encode* and *decode* for lists are

```
instance encode (List a) | encode a where
    encode x = encode (fromList x)
instance decode (List a) | decode a where
    decode x = let(l, x) = decode x in(toList l, x)
```

We can see that arguments are converted with *fromList* and the return values with *toList*. Since *decode* returns a tuple, we need an additional *let* construct. This shows that the conversion not only depends on whether the argument or the result is converted, but on the structure of the overloaded function type. We need to generate such conversions automatically; a solution will be presented in 4.3.

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 [2] for details. Another problem is that we need to provide different mapping functions for different kinds: like *fmap* for kind $\star \to \star$, *bimap* for kind $\star \to \star \to \star$ and so on. Both of these problems are solved by the approach with kind-indexed types [4]. In our design, descrined in the following section, we use this approach in combination with the 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

$$
\begin{array}{ll}
\textbf{generic}\ \mathit{map}\ a_1\ a_2 & ::\ a_1 \rightarrow a_2 \\
\textbf{instance}\ \mathit{map}\ \mathit{Int}\ \textbf{where} & \\
\quad \mathit{map}\ x & = x \\
\textbf{instance}\ \mathit{map}\ \mathit{UNIT}\ \textbf{where} & \\
\quad \mathit{map}\ x & = x \\
\textbf{instance}\ \mathit{map}\ \mathit{PAIR}\ \textbf{where} & \\
\quad \mathit{map}\ \mathit{mapx}\ \mathit{mapy}\ (\mathit{PAIR}\ x\ y) & = \mathit{PAIR}\ (\mathit{mapx}\ x)\ (\mathit{mapy}\ y) \\
\textbf{instance}\ \mathit{map}\ \mathit{EITHER}\ \textbf{where} & \\
\quad \mathit{map}\ \mathit{mapl}\ \mathit{mapr}\ (\mathit{LEFT}\ x) & = \mathit{LEFT}\ (\mathit{mapl}\ x) \\
\quad \mathit{map}\ \mathit{mapl}\ \mathit{mapr}\ (\mathit{RIGHT}\ x) & = \mathit{RIGHT}\ (\mathit{mapr}\ x)
\end{array}
$$

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 $\star \rightarrow \star \rightarrow \star$ needs two additional arguments: one for each type argument. The generic definition can be viewed as a set of classes, one class per kind.

$$
\begin{array}{lll}
\textbf{class}\ \mathit{map}_{\star}\ t & :: & t \rightarrow t \\
\textbf{class}\ \mathit{map}_{\star\rightarrow\star}\ t & :: & (a_1 \rightarrow a_2)\ (t\ a_1) \rightarrow (t\ a_2) \\
\textbf{class}\ \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ t & :: & (a_1 \rightarrow a_2)\ (b_1 \rightarrow b_2)\ (t\ a_1\ b_1) \rightarrow (t\ a_2\ b_2) \\
\ldots & &
\end{array}
$$

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

$$
\begin{array}{ll}
\textbf{instance}\ \mathit{map}_{\star}\ \mathit{Int}\ \textbf{where} & \\
\quad \mathit{map}_{\star}\ x & = x \\
\textbf{instance}\ \mathit{map}_{\star}\ \mathit{UNIT}\ \textbf{where} & \\
\quad \mathit{map}_{\star}\ x & = x \\
\textbf{instance}\ \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ \mathit{PAIR}\ \textbf{where} & \\
\quad \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ \mathit{mapx}\ \mathit{mapy}\ (\mathit{PAIR}\ x\ y) & = \mathit{PAIR}\ (\mathit{mapx}\ x)\ (\mathit{mapy}\ y) \\
\textbf{instance}\ \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ \mathit{EITHER}\ \textbf{where} & \\
\quad \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ \mathit{mapl}\ \mathit{mapr}\ (\mathit{LEFT}\ x) & = \mathit{LEFT}\ (\mathit{mapl}\ x) \\
\quad \mathit{map}_{\star\rightarrow\star\rightarrow\star}\ \mathit{mapl}\ \mathit{mapr}\ (\mathit{RIGHT}\ x) & = \mathit{RIGHT}\ (\mathit{mapr}\ x)
\end{array}
$$

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

The type specified in the **generic** declaration is used to compute the type of mapping for a data type. The type of the generic mapping function is

$$:: \mathit{Map}\ a_1\ a_2 := = a_1 \rightarrow a_2$$

The types of the mapping function for a type of any kind is computed by

$$\begin{array}{ll} :: \mathit{Map}_\star\ t_1\ t_2 & := = \mathit{Map}\ t_1\ t_2 \\ :: \mathit{Map}_{k \rightarrow l}\ t_1\ t_2 & := = \forall a_1\ a_2.\ (\mathit{Map}_k\ a_1\ a_2) \rightarrow \mathit{Map}_l\ (t_1\ a_1)\ (t_2\ a_1) \end{array}$$

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

class $\mathit{map}_k\ t :: \mathit{Map}_k\ t\ t$

The type specified in a generic declaration, like *Map*, is called the poly-kinded type of the generic function. For more details see [4]. 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 $\mathit{Map}_k\ t\ t$ 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* and *Rose* we write:

instance *map List* **generic**
instance *map Rose* **generic**

The mapping for types of kind $\star \rightarrow \star$, like lists and rose trees, can be used as usually, except for kind indexes:

$\mathit{map}\{|\star \rightarrow \star|\}$ *inc (Cons 1 (Cons 2 (Cons 3 Nil)))*
$\mathit{map}\{|\star \rightarrow \star|\}$ *toString (Rose 1 (Cons (Rose 2 Nil) Nil)*

From now on for readability reasons we will write kind indexes as subscripts. The main idea is that from the generic definition of map we get more than just mapping function for types of kind $\star \rightarrow \star$. We get mapping function for types of all (currently first-order) kinds. For example, we can also get the mapping for type *Tree*, which is of kind $\star \rightarrow \star \rightarrow \star$.

instance *map Tree* **generic**

It can be used as in

$\mathit{map}_{\star \rightarrow \star \rightarrow \star}$ *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.

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

generic *eq t* $\quad$ *:: t t* $\rightarrow$ *Bool*
instance *eq Int* **where**
$\quad$ *eq x y* $\quad$ = *eqInt x y*
instance *eq UNIT* **where**
$\quad$ *eq x y* $\quad$ = *True*
instance *eq PAIR* **where**
$\quad$ *eq eqx eqy (PAIR x1 y1) (PAIR x2 y2)*
$\quad\quad$ = *eqx x1 x2* && *eqy y1 y2*
instance *eq EITHER* **where**
$\quad$ *eq eql eqr (LEFT x) (LEFT y)* $\quad$ = *eql x y*
$\quad$ *eq eql eqr (RIGHT x) (RIGHT y)* $\quad$ = *eqr x y*
$\quad$ *eq eql eqr x y* $\quad$ = *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**
instance *eq Rose* **generic**

The equality can be used as in:

$eq_{\star}$ *(Cons 1 (Cons 2 Nil)) (Cons 1 (Cons 2 Nil))*
$eq_{\star}$ *(Rose 1 [Rose 2 []]) (Rose 1 [Rose 2 []])*
$eq_{\star}$ *(Bin 1 (Tip 2) (Tip 3)) (Bin 1 (Tip 2) (Tip 3))*

But the equality defined here is more general then the one defined in Section 2. For instance, when comparing lists one can specify how to compare the elements of the lists.

$eq_{\star\rightarrow\star}$ *(*λ*x y*$\rightarrow$$eq_{\star}$ *(length x) (length y)) [[1,2],[3,4]] [[1,1], [2,2]]* $\Rightarrow$ *True*

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

Now the question is, which kind indexes are applicable? For a type $T\ a_1\ \ldots\ a_n$ of kind k the only applicable kind indexes are k and $\star$. For example, type *Tree* has kind $\star \rightarrow \star \rightarrow \star$. The kind $\star \rightarrow \star \rightarrow \star$ is applicable because we have the instance

instance $eq_{\star\rightarrow\star\rightarrow\star}$ *Tree* **generic**

The instance for kind $\star$ is additionally generated to make simple comparison of trees possible:

instance $eq_\star$ *(Tree a b)* | $eq\ a$ & $eq\ b$ **where**
    $eq_\star\ x\ y = eq_{\ \star\to\star\to\star}\ eq_\star\ eq_\star\ x\ y$

One can define the standard equality operator using the generic equality.

(==) **infixr** $5 :: t\ t \to Bool \mid eq_\star\ t$
(==) $x\ y = eq_\star\ x\ y$

The user can provide his own instances of a generic function instead of making the compiler to generate them.

Consider an application of *map*

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

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

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

To make such applications simpler in the future we may allow type indexes, as in Generic Haskell:

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

Note that $map_{[[a]]}$ is not overloaded, whereas $map_{\star\to\star}$ is.

As one more example we show the right reduce function, which is a generalization of *foldr* on lists. It takes a structure a and a value of type b and collapses it 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* $a :: a\ b \to b$
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 "zero". The instance for pairs uses the result of reduction for the second element of the pair as the "zero" 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* on types of kind $\star \to \star$ using *rreduce*.

$foldr :: (a\ b \to b)\ b\ (t\ a) \to b \mid rreduce_{\star\to\star}\ t$
$foldr\ op\ e\ x = rreduce_{\star\to\star}\ op\ x\ e$

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:

$foldr :: (a \to b \to b)\ b\ [a] \to b$

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

$foldr :: (a \to b \to b)\ b\ (t\ a) \to b$

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

$$
\begin{array}{lll}
\textbf{class}\ rreduce_{\star}\ a & :: & a\ b \to b \\
\textbf{class}\ rreduce_{\star\to\star}\ a & :: & (a_1 \to b \to b)\ (a\ a_1)\ b \to b \\
\textbf{class}\ rreduce_{\star\to\star\to\star}\ a & :: & (a_1 \to b \to b)\ (a_2 \to b \to b)\ (a\ a_1\ a_2)\ b \to b
\end{array}
$$

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 non-generic Clean.

4.1 Classes and instances

In general, a generic definition looks like

$\textbf{generic}\ g\ a_1\ \ldots\ a_n :: G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m$

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). The polykinded type is used to compute the type of the function for a kind:

$$
\begin{array}{lll}
G_{\star}\quad t_1\ \ldots\ t_n\ p_1\ \ldots\ p_m & = & G\ t_1\ \ldots\ t_n\ p_1\ \ldots\ p_m \\
G_{k\to l}\quad t_1\ \ldots\ t_n\ p_1\ \ldots\ p_m & = & \forall a_1 \ldots a_n . (G_k\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m) \\
 & & \to G_l\ (t_1\ a_1)\ \ldots\ (t_n\ a_n)\ p_1\ \ldots\ p_m
\end{array}
$$

The generic extension translates such 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:

$\textbf{class}\ g_k\ t :: G_k\ t\ \ldots\ t\ p_1\ \ldots\ p_m$

Each instance of a generic function is bound to one of the classes according to the kind of the instance type.

4.2 Generic type representation

To specialize a generic function to a data type one needs the generic representation of that type. The need for generic type representation has the following aspects:

- The generic representation type itself. We denote the generic representation of type T with T°. As already mentioned, we use a binary representation of sums and products.
- The conversion functions from T to T° and back.
- A generic function g of type $G\ a_1\ \ldots\ a_n$ can be specialized to the generic representation T° as shown in section 2. But we need it to be specialized to type T. Thus, we need a conversion function from $G\,T^\circ\ \ldots\ T^\circ$ to $G\,T\ \ldots\ T$.

The algorithms of building the generic representation types and the conversion functions are described by Hinze in [5]. The conversion functions are packed into a structure defined in the generic prelude:

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

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

$$
\begin{array}{lll}
\mathit{iso}_{List} & \multicolumn{2}{l}{:: \mathit{Iso}\ (\mathit{List}\ a)\ (\mathit{List}^\circ\ a)} \\
\mathit{iso}_{List} & \multicolumn{2}{l}{= \{\mathit{iso}{=}\mathit{iso}, \mathit{osi}{=}\mathit{osi}\}} \\
\textbf{where} & \mathit{iso}\ \mathit{Nil} & = \mathit{LEFT}\ \mathit{UNIT} \\
 & \mathit{iso}\ (\mathit{Cons}\ x\ xs) & = \mathit{RIGHT}\ (\mathit{PAIR}\ x\ xs) \\
 & \mathit{osi}\ (\mathit{LEFT}\ \mathit{UNIT}) & = \mathit{Nil} \\
 & \mathit{osi}\ (\mathit{RIGHT}\ (\mathit{PAIR}\ x\ xs)) & = \mathit{Cons}\ x\ xs
\end{array}
$$

4.3 Bidirectional mapping

In section 2 we showed how to define a generic function for a data type in terms of the generic representation of that type. We use the instance for the generic representation of a type to define the instance for the type itself. To do that we need to convert the arguments and the results to and from the generic representation. We noted that this conversion depends on the type of the generic function. In this subsection we show how to cope with this issue.

To deal with the conversion we use bidirectional mapping functions, as in [3]. The need to have conversion in both directions comes from the fact that the arrow type is contravariant in the argument position. The generic function types and the data types in general may involve arrows. We define the bidirectional mapping for a generic function type by induction on the structure of types. Type terms are formed according to

$$
\begin{array}{lll}
t & = T & \textit{type constructor} \\
 & \mid\ a_i & \textit{polykinded type variable} \\
 & \mid\ p_i & \textit{polymorphic type variable} \\
 & \mid\ t\ s & \textit{type application} \\
 & \mid\ t \to s & \textit{arrow type}
\end{array}
$$

The bidirectional mapping for a generic function g of type $G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m$ is a function

$$\begin{array}{l} \mathit{bmap}_g :: (\mathit{Iso}\ a_1\ a_1^\circ)\ \ldots\ (\mathit{Iso}\ a_n\ a_n^\circ) \to \\ \qquad (\mathit{Iso}\ (G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m)\ \ldots\ (G\ a_1^\circ\ \ldots\ a_n^\circ\ p_1\ \ldots\ p_m)) \\ \mathit{bmap}_g\ v_1\ \ldots\ v_n = \mathit{bmap}(G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m) \end{array}$$

The function lifts the isomorphisms for the arguments to the isomorphism for the function type. The right hand side is defined by induction on the structure of type $G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m$:

$$\begin{array}{ll} \mathit{bmap}(T) & = \mathit{bmap}_T \\ \mathit{bmap}(a_i) & = v_i \\ \mathit{bmap}(p_i) & = \mathit{bmapId} \\ \mathit{bmap}(t\ s) & = \mathit{bmap}(t)\ \mathit{bmap}(s) \\ \mathit{bmap}(t \to s) & = \mathit{bmap}_\to\ \mathit{bmap}(t)\ \mathit{bmap}(s) \end{array}$$

Here v_i is a term variable corresponding to the type variable a_i. Currently polymorphic type variables p_i in polykinded types are limited to be of kind $\star$; their mapping is the identity mapping $\mathit{bmapId} = \{\mathit{iso}{=}\mathit{id},\ \mathit{osi}{=}\mathit{id}\}$.

For a data type

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

the bidirectional mapping is

$$\begin{array}{l} \mathit{bmap}_T\ v_1\ \ldots\ v_n = \{\mathit{iso}{=}\mathit{iso}, \mathit{osi}{=}\mathit{osi}\} \\ \textbf{where} \\ \quad \mathit{iso}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) = K_1\ (\mathit{bmap}(t_{11}).\mathit{iso}\ x_{11})\ \ldots\ (\mathit{bmap}(t_{1l_1}).\mathit{iso}\ x_{1l_1}) \\ \quad \ldots \\ \quad \mathit{iso}\ (K_m\ x_{m1}\ \ldots\ x_{ml_m}) = K_m\ (\mathit{bmap}(t_{m1}).\mathit{iso}\ x_{m1}) \ldots (\mathit{bmap}(t_{ml_m}).\mathit{iso}\ x_{ml_m}) \\ \quad \mathit{osi}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) = K_1\ (\mathit{bmap}(t_{11}).\mathit{osi}\ x_{11})\ \ldots\ (\mathit{bmap}(t_{1l_1}).\mathit{osi}\ x_{1l_1}) \\ \quad \ldots \\ \quad \mathit{osi}\ (K_m\ x_{n1}\ \ldots\ x_{nm_n}) = K_m\ (\mathit{bmap}(t_{m1}).\mathit{osi}\ x_{m1}) \ldots (\mathit{bmap}(t_{ml_m}).\mathit{osi}\ x_{ml_m}) \end{array}$$

where $x_{ij} :: t_{ij}$ is the jth argument of the data constructor K_i. For instance, bidirectional mapping for the list type is

$$\begin{array}{l} \mathit{bmap}_{List}\ v = \{\mathit{iso}{=}\mathit{iso}, \mathit{osi}{=}\mathit{osi}\} \\ \textbf{where} \\ \qquad \mathit{iso}\ \mathit{Nil} = \mathit{Nil} \\ \qquad \mathit{iso}\ (\mathit{Cons}\ x\ xs) = \mathit{Cons}\ (v.\mathit{iso}\ x)\ ((\mathit{bmap}_{List}\ v).\mathit{iso}\ xs) \\ \qquad \mathit{osi}\ \mathit{Nil} = \mathit{Nil} \\ \qquad \mathit{osi}\ (\mathit{Cons}\ x\ xs) = \mathit{Cons}\ (v.\mathit{osi}\ x)\ ((\mathit{bmap}_{List}\ v).\mathit{osi}\ xs) \end{array}$$

The bidirectional mapping for the arrow type is

$$\begin{array}{l} \mathit{bmap}_\to\ \mathit{bmaparg}\ \mathit{bmapres} = \{\mathit{iso}{=}\mathit{iso},\ \mathit{osi}{=}\mathit{osi}\} \\ \textbf{where} \\ \qquad \mathit{iso}\ f = \mathit{bmapres.iso} \cdot f \cdot \mathit{bmaparg.osi} \\ \qquad \mathit{osi}\ f = \mathit{bmapres.osi} \cdot f \cdot \mathit{bmaparg.iso} \end{array}$$

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

The examples of the functions generated for the equality and the mapping are

$$
\begin{aligned}
&bmap_{\mathrm{eq}} :: (Iso\ a\ a^{\circ}) \to (Iso\ (Eq\ a)\ (Eq\ a^{\circ}))\\
&bmap_{\mathrm{eq}}\ v = bmap_{\to}\ v\ (bmap_{\to}\ v\ bmap_{\mathrm{Bool}})\\
&bmap_{\mathrm{map}} :: (Iso\ a_1\ a_1^{\circ})\ (Iso\ a_2\ a_2^{\circ}) \to (Iso\ (Map\ a_1\ a_2)\ (Map\ a_1^{\circ}\ a_2^{\circ}))\\
&bmap_{\mathrm{map}}\ v_1\ v_2 = bmap_{\to}\ v_1\ v_2
\end{aligned}
$$

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

4.4 Generated instances

To specialize a generic function g with a polykinded type $G\ a_1\ \ldots\ a_n\ p_1\ \ldots\ p_m$ to a type T of kind k the compiler generates the following:

$$
\begin{aligned}
&g_T :: G_k\ T\ \ldots\ T\ p_1\ \ldots\ p_m\\
&g_T = (bmap_g\ iso_T\ \ldots\ iso_T).osi\ g_{T^{\circ}}\\
&\mathbf{instance}\ g_k\ T\ \mathbf{where}\ g_k = g_T
\end{aligned}
$$

The function $g_{T^{\circ}}$ is generated by interpreting the type T°: type application is replaced by value application, type abstraction by value abstraction and so on. For example, consider the generic equality

$$
\begin{aligned}
&:: Eq\ a :== a \to a \to Bool\\
&\mathbf{generic}\ eq :: Eq\ a
\end{aligned}
$$

For lists we have

$$
\begin{aligned}
&eq_{\mathrm{List}^{\circ}} :: (Eq\ a) \to Eq\ (List^{\circ}\ a)\\
&eq_{\mathrm{List}^{\circ}}\ eqa = eq_{\mathrm{EITHER}}\ eq_{\mathrm{UNIT}}\ (eq_{\mathrm{PAIR}}\ eqa\ (eq_{\mathrm{List}}\ eqa))\\
&eq_{\mathrm{List}} :: ((Eq\ a) \to Eq\ (List\ a))\\
&eq_{\mathrm{List}} = (bmap_{\mathrm{eq}}\ iso_{\mathrm{List}}).osi\ eq_{\mathrm{List}^{\circ}}\\
&\mathbf{instance}\ eq_{\star\to\star}\ List\ \mathbf{where}\ eq_{\star\to\star} = eq_{\mathrm{List}}
\end{aligned}
$$

In [4] Hinze proves that the functions specialized in this way are well-typed.

Additionally we create instances for kind $\star$. For each instance on a type of a kind other then $\star$ a shortcut instance for kind $\star$ is created. Consider the instance of a generic function g for a type $T\ a_1\ \ldots\ a_n$, $n \geq 1$. The kind k of the type T is $k = k_1 \to \ldots \to k_n \to \star$.

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

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

instance $eq_\star$ *[a]* | $eq_\star$ *a* **where**
  $eq_\star\ x\ y = eq_{\star\to\star}\ eq_\star\ x\ y$
instance $eq_\star$ *Tree a b* | $eq_\star$ *a* & $eq_\star$ *b* **where**
  $eq_\star\ x\ y = eq_{\star\to\star\to\star}\ eq_\star\ eq_\star\ x\ y$

These instances make it 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]*

Note that kind $\star$ instances turn explicit arguments into dictionaries of the overloading system.

5 Customized instances

In this section we present an extension that allows for customization of generated instances. Let's consider a data type for terms in a compiler as an example.

```
:: 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 containing expressions:

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

*UNIT*s and *Int*s do not contain variables, so the instances return empty lists. The instance on *Var* returns the variable as a singleton list. For pairs the variables are collected in both components; the concatenated list is returned. For sums the variables are collected in the arguments. 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.

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. In a real-world compiler the type *Expr* may contain many alternatives. It is tedious to provide all the alternatives in each generic function, even if only a couple of them are essential. If the type is modified all such places have to be modified as well. The generic extension provides a way to deal with this problem. The user can refer to the generic implementation of an instance that he or she provides.

In the example the instance on *Expr* can be written more compactly:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{fvs}\ \mathit{Expr}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{fvs}\ (\mathit{ELambda}\ \mathit{var}\ \mathit{expr}) & = \mathit{filter}\ ((<>)\ \mathit{var})\ (\mathit{fvs}_\star\ \mathit{expr}) \\
\mathit{fvs}\ x & = \mathit{fvs}\{|\textbf{generic}|\}\ x
\end{array}
\end{array}
$$

The name $\mathit{fvs}\{|\textbf{generic}|\}$ is bound to a function that is the generic implementation of the instance where it is defined. The code generated for the instance on Expr is:

$$
\begin{array}{ll}
\mathit{fvs}^g_{Expr}\ x & = (\mathit{bmap}_{fvs}\ \mathit{iso}_E\mathit{xpr}).\mathit{osi}\ \mathit{fvs}_{Expr^\circ}\ x \\
\mathit{fvs}_{Expr}\ (\mathit{ELambda}\ \mathit{var}\ \mathit{expr}) & = \mathit{filter}\ ((<>)\ \mathit{var})\ (\mathit{fvs}_{Expr}\ \mathit{expr}) \\
\mathit{fvs}_{Expr}\ x & = \mathit{fvs}^g_{Expr}\ x
\end{array}
$$

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

Another example is the mapping function for expressions

$$
\begin{array}{l}
\textbf{generic}\ \mathit{mapExpr}\ a :: (\mathit{Expr} \to \mathit{Expr})\ a \to a \\
\textbf{instance}\ \mathit{mapExpr}\ \mathit{UNIT}\ \textbf{where} \\
\quad \mathit{mapExpr}\ f\ \mathit{UNIT} = \mathit{UNIT} \\
\textbf{instance}\ \mathit{mapExpr}\ \mathit{Int}\ \textbf{where} \\
\quad \mathit{mapExpr}\ f\ x = x \\
\textbf{instance}\ \mathit{mapExpr}\ \mathit{PAIR}\ \textbf{where} \\
\quad \mathit{mapExpr}\ \mathit{fx}\ \mathit{fy}\ f\ (\mathit{PAIR}\ x\ y) = \mathit{PAIR}\ (\mathit{fx}\ f\ x)\ (\mathit{fy}\ f\ y) \\
\textbf{instance}\ \mathit{mapExpr}\ \mathit{EITHER}\ \textbf{where} \\
\quad \mathit{mapExpr}\ \mathit{fl}\ \mathit{fr}\ f\ (\mathit{LEFT}\ x) = \mathit{LEFT}\ (\mathit{fl}\ f\ x) \\
\quad \mathit{mapExpr}\ \mathit{fl}\ \mathit{fr}\ f\ (\mathit{RIGHT}\ x) = \mathit{RIGHT}\ (\mathit{fr}\ f\ x) \\
\textbf{instance}\ \mathit{mapExpr}\ \mathit{Expr}\ \textbf{where} \\
\quad \mathit{mapExpr}\ f\ x = f\ (\mathit{mapExpr}\{|\textbf{generic}|\}\ f\ x)
\end{array}
$$

All the instances are similar to the instance of *map*, except for the instance on *Expr*, which first maps the sub-expressions and then applies the argument function to the resulting expression. Analogously, a mapping function for a system of mutually recursive types can be defined; a function (like $\mathit{Expr} \to \mathit{Expr}$) is needed for each type in the system.

With the mapping function we can define a function for constant function elimination.

$cfe :: a \to a \mid mapExpr_\star\ a$
$cfe\ x = mapExpr_\star\ elim\ x$
where $elim\ (EApp\ (ELambda\ v\ body)\ arg)$
$\qquad \mid not\ (isMember\ v\ (fvs_\star\ body)) \quad = body$
$\quad elim\ expr \qquad\qquad\qquad\qquad\qquad = expr$

Note the separation of concerns: the mapping function walks the recursive structure of the expression, the *cfe* function involves only constructors it directly works on.

To generalize *fvs* one would like to implement a generic fold function, in a way similar to map. Unfortunately, the separation of concerns cannot be done for fold: explicit recursion is required. Another disadvantage is that the user has to provide similar functions for different term types (e.g. for mapping type terms). See also the discussion of "large bananas" [9] in section 7).

6 The module system

Generic definitions and generic instances are exported and imported in Clean in the same way as classes and class instances. The Clean module system has separate definition and implementation modules. Each logical module consists of a definition and implementation module with the same name. Symbols defined only in the implementation module are local to the module. Symbols defined in the definition module are exported and can be used in other modules.

The generic extension cannot generate instances for data types that are abstract in the module being compiled. It issues an error message. To generate such an instance one would need to know the right hand side of the abstract type. The generated code would depend on the definition of an abstract type. So, a generic instance for an abstract type can be defined only in the module, where the abstract type itself is defined and it's implementation is known. The instance can be exported from this module together with the abstract type.

We illustrate the idea by example. Generic equality is defined in module *equality*. The export declaration is in the definition module *equality*. The implementation of *eq* resides in the implementation module that is not shown.

definition module *equality*
generic $eq\ t :: t\ t \to Bool$
instance *eq Int, UNIT, PAIR, EITHER*

Definition module *stack* exports abstract type Stack and an instance of equality on stacks.

definition module *stack*
import *equality*
:: Stack a
instance *eq Stack*

Implementation module *stack* contains the definition of the abstract type. The generic instance can be successfully generated because the right hand side of the type is available.

implementation module *stack*
import *equality*
:: Stack a :== [a]
instance *eq Stack* **generic**

A client module can call the equality function to compare abstract stacks.

Instance code for generic function g on type T is generated as a part of the module, where the **instance** g T **generic** clause is specified. It means that if, for instance, in two modules an equality for lists is asked to be generated, each module will get a copy of the code. This design can lead to code growth. As a solution to the problem, one can implement a cash of generated instances, so that the code is shared.

Currently in Clean an instance body cannot be defined in a definition module. It means that the instance bodies are not available to the compiler when a client module is compiled. For instance, in the example above the body of *eq UNIT* defined in the module *equality* is not known in the module *stack*. As a consequence, the equality for units cannot be inlined. This is especially important for generics because additional complexity introduced by the generic type representation cannot be eliminated. In the future we plan to solve the problem by allowing instance definitions in definition modules, so that they will be available to the optimizer.

7 Related Work

7.1 Generic Haskell Prototype

Generic Haskell is a generic extension for Haskell. The design is based on kind-indexed types, described in [4]. The Generic Haskell compiler takes Generic Haskell code and produces Haskell code as output. As opposed, generic extension of Clean is a part of the Clean compiler. It makes the design and the implementation easier, because we can piggy-back on other features of the language. In particular, we do not need to change the type system of the language.

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*:

(==) **infixr** $5 :: t\ t \rightarrow Bool \mid eq_\star\ t$
$(==)\ x\ y = eq_\star\ x\ y$

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.

Generic Haskell supports generic instances for higher-order kinded types. It requires rank-2 type signatures and local quantification in data types. Clean currently does not support these features, but we are busy implementing them.

7.2 Generics in Glasgow Haskell Compiler

The design of Glasgow Haskell's generic extension is described in [3]. In GHC generic definitions are used to define default implementation of class members. The compiler generates the member bodies using the generic definitions. A class can have any number of generic members. The design gives a systematic meaning to Haskell's **deriving** construct. Default methods can be derived for type classes whose class argument is of kind $\star$. Other kinds are not supported. That means that functions like mapping cannot be defined generically. See also [2].

Generic Clean supports a special **generic** construct that gives birth to a kind-indexed family of classes. Each class has only one member. The generic definition provides default implementation for members of all of these classes. For instance, it possible in Clean to customize how elements of lists are compared:

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

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 first-order) kinds. This is especially important for functions like mapping.

7.3 Dictionary Passing for Polytypic Polymorphism in SML/NJ

In [6] Chen and W. Appel describe an approach to implement specialization of generic functions using dictionary passing. In Clean the generic extension is based on type classes, which are implemented using dictionaries. The SML/NJ extension requires extension of the kind system of the language. The kind of a type indicates which generic functions are applicable for this type. The kind determines the type of the dictionary for a data type. In Clean we do not need to modify the kind system of the language. The dictionaries are created by the overloading system as usually.

7.4 PolyP

PolyP [7] 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 and anamorphisms can be defined. This is possible due to explicit recursion, which causes limitation to regular types. It remains to be seen how to support such recursion schemes in Clean.

7.5 Dealing with Large Bananas

In [9] 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 [9]). To support type-unifying algebras (fold-like) we need explicit recursion.

8 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 [4] 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.

The main problems we currently are working on and planning to work on in the near future:

- *Types of higher-order kinds.* Generic Clean does not fully support types of higher-order kinds. We are busy adding it to the compiler.
- *Support for explicit recursion on types.* As noted in section 7 our design does not support recursion schemes like catamorphisms. We plan to add the support in the future.
- *Support for constructor information.* In order to implement pretty printers and parsers one needs information about the data constructors: names, arities and so on. The design described here does not keep the constructor names in the generic representation. The support can be implemented in the style described in [3].
- *Uniqueness typing.* In this paper we ignored the uniqueness typing of the language. Though we have a working prototype of the uniqueness in polykinded types, we are busy formalizing it.

– *Optimization of the generated code.* Currently our prototype lacks optimization of the generated code. We are convinced that a partial evaluator can optimize out the conversion code introduced by the translation of generics. Our group is working on such an optimizer.
– *Class contexts in generic types.* The current design does not support context restrictions on neither generic nor non-generic variables in the polykinded types of the generic functions. Currently one also cannot define a generic function using other generic functions, in the way it is done with overloaded functions.
– *Array types.* The generic extension cannot generate instances for array types.

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. We would like to thank Ralf Hinze for the discussion of customized instances.

References

1. Ralf Hinze. *Polytypic programming with ease.* Technical Report IAI-TR-99-2, Institut für Informatik III, Universität Bonn, February 1999
2. 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.
3. Ralf Hinze and Simon Peyton Jones. *Derivable Type Classes.* In Graham Hutton, editor, Proceedings of the Fourth Haskell Workshop, Montreal, Canada, September 17, 2000
4. 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.
5. 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.
6. Juan Chen and Andrew W. Appel. *Dictionary Passing for Polytypic Polymorphism* Princeton University Computer Science TR-635-01, March 2001.
7. 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.
8. 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
9. 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.