A Generic Programming Extension for Haskell

Ralf Hinze

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany

ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Abstract

Many functions can be defined completely generically for all datatypes. Examples include pretty printers (eg `show`), parsers (eg `read`), data converters, equality and comparison functions, mapping functions, and so forth. This paper proposes a generic programming extension that enables the user to define such functions in Haskell. In particular, the proposal aims at generalizing Haskell's `deriving` construct, which is commonly considered deficient since instance declarations can only be derived for a few predefined classes. Using generic definitions derived instances can be specified for arbitrary user-defined type classes and for classes that abstract over type constructors of first-order kind.

1 Introduction

Generic or polytypic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different datatypes. Typical examples for so-called generic functions include pretty printers (eg `show`), parsers (eg `read`), functions that convert data into a universal datatype (eg bit strings, character strings, XML documents), equality and comparison functions, mapping functions (`map`), and so forth. This paper proposes a generic programming extension for Haskell that enables the user to define such functions completely generically for all datatypes, including nested datatypes, mutually recursive datatypes, and datatypes involving function spaces. The central idea is to parameterize a function definition with a type or a type constructor and to define the function by induction on the structure of types or type constructors.

Interestingly, Haskell already provides a rudimentary form of genericity. Its derivable class methods can be seen as simple examples for generic functions. Using the `deriving` construct instance declarations (ie collections of class methods) can be automatically generated for user-defined datatypes. Every Haskell programmer probably uses this mechanism to derive, for instance, `Eq` or `Show` instances (ie `(==)` and `show` methods) for newly defined datatypes. As matters stand, the `deriving` construct suffers from several drawbacks. First of all, the definitions of the generic functions are hard-wired into the various Haskell compilers. This implies, in particular, that libraries that contain derivable classes (such as `Ix`) have a special status since the Haskell compiler must know details of their code. Clearly, this is an undesirable situation. Furthermore, derived instances are only informally defined in the Haskell Report [19, Appendix D]. For instance, Section D.4 of the Report contains the following statement.

> "A precise definition of the derived `Read` and `Show` instances for general types is beyond the scope of this report."

Second, instances can only be derived for a few predefined classes. The user can neither modify existing derivable classes (for instance, to adapt `show` to her needs) nor can she define derivable classes herself. Finally, `deriving` does not work for all user-defined types—a fact that is not very well known. For instance, if a datatype is parameterized by a type constructor as in

```
data Twice m v =  Twice (m (m v))
```

then deriving `Eq` or `Show` fails in general. The reason for this failure, which is quite fundamental, is explained in Section 3.1.

If we augment Haskell by generic definitions, all of these problems can be overcome. The meaning of derived instances can be made precise (Appendix B, for instance, precisely defines the generic `show` method), the user can define her own derivable classes, and instances can be derived for all datatypes definable in Haskell.

In the rest of this introduction let us take a brief look at generic definitions. We have already mentioned the basic idea that a generic function is parameterized by datatype and is defined by induction on the structure of types. Now, the types introduced by Haskell's `data` construct are essentially sums of products. The components of a product may depend on other user-defined datatypes or on primitive types such as `Char`, `Int`, or '`->`'. As an example, consider the datatype of polymorphic binary trees (the constructor `Bin` contains an additional size field).

```
data Tree a = Tip | Bin Int (Tree a) a (Tree a)
```

Let us make the structure of this definition more explicit by writing '`+`' for sums and '`*`' for products.

```
Tree a = 1  +  Int * Tree a * a * Tree a
```

Here `1` is an abbreviation for the unit type. The equation makes explicit that `Tree` is a binary sum; the first component of the sum is a 0-tuple and the second is a 4-tuple, which can be represented by a nested binary product. Likewise, all datatypes can be decomposed into binary sums, binary products, and primitive types. Consequently, to define a function generically for all datatypes it suffices to specify its action on binary sums, binary products, and primitive types. As an example, the following equations define the generic function `encode(t)`, which converts elements of type `t` into bit strings implementing a simple form of data compression.

```
data Bit                  =  O | I

encode(t)                 :: t -> [Bit]
encode(1)     ()          =  []
encode(Char)  x           =  encodeChar x
encode(Int)   x           =  encodeInt  x
encode(a+b)   (Left  x)   =  O : encode(a) x
encode(a+b)   (Right y)   =  I : encode(b) y
encode(a*b)   (x, y)      =  encode(a) x
                          ++ encode(b) y
```

The first parameter of `encode` is its type argument. The type signature makes explicit that the type of `encode(t)` depends on the type argument `t`. The definition proceeds by case distinction on `t`: we have one equation for `1`, `Char`, `Int`, and '`*`' and two equations for '`+`'. Note that `1`, '`+`', and '`*`' serve as abbreviations for the predefined types `()`, `Either`, and `(,)` which are given by the following definitions.

```
data ()          =  ()
data Either a b  =  Left a | Right b
data (a, b)      =  (a, b)
```

Let us consider each equation of `encode(t)` in turn. To encode the single element of `1` no bits are required. Characters and integers are encoded using the primitive functions `encodeChar` and `encodeInt`, whose existence we assume. To encode an element of a sum we emit one bit for the constructor followed by the encoding of its argument. Finally, the encoding of a pair is given by the concatenation of the component's encodings. Given this definition we can compress elements of arbitrary datatypes. For instance, `encode(Tree Char)` of type `Tree Char -> [Bit]` compresses binary trees with character labels.

The rest of this paper is organized as follows. Section 2 introduces the various forms of generic definitions and sketches their semantics. Section 3 explains how to blend generic definitions with Haskell's type classes. In particular, we show how to define derivable type classes. Section 4 presents several examples for generic functions. Section 5 reviews related work and Section 6 concludes. Appendix A details the necessary changes to the syntax and Appendix B lists the modifications to the Standard Prelude and to the Standard Libraries. Note that this paper describes the generic programming extension primarily from the user's point of view. The theoretical background of this extension is described in a companion paper [7]. Furthermore, note that the extension has not been implemented yet but we plan to do so in the near future.

2 Generic definitions

2.1 Type-indexed values

A generic function—or rather, a generic value is either defined inductively or in terms of other generic values. Here is the prototype for an inductive definition.

```
g(t)     :: ... t ...
g(1)     =  ...
g(Char)  =  ...
g(Int)   =  ...
...
g(a+b)   =  ... g(a) ... g(b) ...
g(a*b)   =  ... g(a) ... g(b) ...
g(a->b)  =  ... g(a) ... g(b) ...
...
```

Type signatures are mandatory for generic values. In fact, the type argument of `g`, which appears on the left-hand side of the signature, identifies `g` as being indexed by type. The equations for `g` employ pattern matching on types. The type patterns on the left-hand sides are restricted to type expressions of the form $\mathtt{T}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_k$, ie a type constructor is applied to type variables $\mathtt{a}_1, \ldots, \mathtt{a}_k$, which have to be distinct. Since `g` is inductively defined, the recursive calls on the corresponding right-hand sides must have the form `g(a)` where `a` is one of the type variables $\mathtt{a}_1, \ldots, \mathtt{a}_k$.

Generic values are usually *not* defined for all types. For instance, `encode(t)` cannot compress functions or floating-point numbers. To be exhaustive a generic definition must include cases for `1`, '`+`', and '`*`'—these cases cover `data` declarations—and additionally for all primitive types that are not defined via a `data` declaration. All in all, there are a dozen primitive types: the Standard Prelude defines `a->b`, `Char`, `Double`, `Float`, `Int`, `Integer`, `IO a`, and `IOError` and the Standard Libraries provide `Array a b`, `Permissions`, `ClockTime`, and `StdGen`.

A generic value is instantiated or applied simply by calling it with an appropriate type argument, which may be an arbitrary type expression. We only require the type expression to be closed, ie it may not contain any type variables.

Generic values can be defined in terms of other generic values. As an example, consider the function `dump(t)`, which writes elements of type `t` in binary format to a given file (`toChar` maps a 16-bit value to a character and `rsplit n` splits its list argument into a list of lists, each of which has length `n`—only the last element may have a length smaller than `n`).

```
dump(t)          :: FilePath -> t -> IO ()
dump(t) fpath x =  writeFile fpath
                $  map toChar
                $  rsplit 16
                $  encode(t) x
```

Note that the type signature is again mandatory and that the type parameter of `dump` must not be omitted. Since type arguments are in general needed for disambiguation—Section 2.6 provides an example—we follow the design principle that value-type dependencies must be made explicit. In general, a non-inductive definition has the following form.

```
g(t) :: ... t ...
g(t) =  ... h(... t ...) ... i(... t ...) ...
```

Here `t` is a type variable. The right-hand side of the definition may involve calls of other generic values. Their type arguments may contain the type variable `t` but no other type variables.

2.2 Semantics

This section gives a brief account of the semantics of generic definitions. First of all, note that generic definitions operate on *binary* sums and products. This is a design decision dictated by pragmatic concerns. We could have based generic definitions on *n-ary* sums and products fitting them more closely to datatype declarations. It appears, however, that

n-ary type constructors are awkward to work with. Therefore, we must make precise how n-ary sums and products are translated into binary ones.

Roughly speaking, the right-hand side of a datatype declaration, $c_1 \mid \cdots \mid c_n$, is translated to the type expression $\Sigma(c_1 \dots c_n)$ defined as follows.

$$\Sigma(t_1 \dots t_n) = \begin{cases} t_1 & \text{if } n = 1 \\ \Sigma(t_1 \dots t_{\lfloor n/2 \rfloor}) + \Sigma(t_{\lfloor n/2 \rfloor+1} \dots t_n) & \text{if } n > 1 \end{cases}$$

The translation essentially creates a balanced expression tree. For instance, $c_1 \mid c_2 \mid c_3 \mid c_4$ is mapped to $(c_1 + c_2) + (c_3 + c_4)$. All generic definitions in this paper with the notable exception of `encode(t)` and `decode(t)` (see Section 4) are insensitive to the translation of n-ary sums. For `encode(t)` this translation scheme is a sensible choice (an unbalanced or a list-like scheme would aggravate the compression rate).

Likewise, the components of a constructor, $C\ t_1\ \dots\ t_k$, are translated to $\Pi(t_1 \dots t_k)$ given by

$$\Pi(t_1 \dots t_n) = \begin{cases} 1 & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ \Pi(t_1 \dots t_{\lfloor n/2 \rfloor}) * \Pi(t_{\lfloor n/2 \rfloor+1} \dots t_n) & \text{if } n > 1 \end{cases} .$$

The generic values defined in this paper happen to be insensitive to the encoding of n-ary products. It is, however, not difficult to define a function that yields different results for different translation schemes. Figure 1 contains the complete translation of `data` and `newtype` declarations. Note that the resulting type expressions include constructor names and record labels. Section 2.3 explains why this is useful. Strictness annotations are, however, ignored. Whether this is a viable choice remains to be seen.

Now, the central idea of the semantics is that type expressions can be interpreted in a fairly straightforward manner as values. To exemplify, the equation for `Tree`

```
Tree a = Tip 1 + Bin ((Int*Tree a) * (a*Tree a))
```

can be seen as defining a one-argument function, which maps type expressions to type expressions. In general, we interpret primitive types and type constructors (such as `1`, '`+`', '`*`', `Char`, `Int`, '`->`' etc) as *value constructors* and types that are defined by datatype declarations as *functions*. Since `Tree` is recursively defined, the resulting type expression is, of course, infinite. In this particular case the type expression can be represented by a circular structure but this does not hold in general, see [8]. Fortunately, Haskell is a non-strict language and is able to generate and to process infinite data structures with ease. Consequently, we may interpret generic values as 'ordinary' functions, which operate on possibly infinite type expressions. It is not possible, however, to specify this semantics via a translation into Haskell. The reason is simply that Haskell's type system is not expressive enough. To specify the type of generic values dependent types are required.

We have seen that datatypes are translated into binary sums and products. This translation must, of course, be duplicated on the value level. For instance, the argument `x` in `encode(Tree Char) x` must be converted to $[\![\mathtt{x}]\!]$ given by

$$\begin{aligned} [\![\mathtt{Tip}]\!] &= \mathtt{Left}\ \ \mathtt{()} \\ [\![\mathtt{Bin}\ e_1\ e_2\ e_3\ e_4]\!] &= \mathtt{Right}\ \ \mathtt{((}[\![e_1]\!]\mathtt{,}[\![e_2]\!]\mathtt{),(}[\![e_3]\!]\mathtt{,}[\![e_4]\!]\mathtt{))}\ . \end{aligned}$$

In general, a constructor application $K_i\ e_1 \dots e_{k_i}$, where K_i is introduced via the declaration

$$\mathtt{data}\ T\ a_1 \dots a_m \ = \ K_1\ t_{11} \dots t_{1k_1} \mid \cdots \mid K_n\ t_{n1} \dots t_{nk_n}\ ,$$

is translated as follows

$$[\![K_i\ e_1 \dots e_{k_i}]\!] \ = \ \sigma_i^n(\pi([\![e_1]\!] \dots [\![e_{k_i}]\!]))\ .$$

The function σ_i^n yields the binary encoding of the constructor K_i.

$$\sigma_i^n(v) = \begin{cases} v & \text{if } n = 1 \\ \mathtt{Left}\ (\sigma_i^{\lfloor n/2 \rfloor}(v)) & \text{if } i \leqslant \lfloor n/2 \rfloor \\ \mathtt{Right}\ (\sigma_{i-\lfloor n/2 \rfloor}^{n-\lfloor n/2 \rfloor}(v)) & \text{otherwise} \end{cases}$$

The definition may look daunting but it has a simple interpretation. If we draw the binary sum as a tree, then σ_i^n specifies the path from the root to the i-th leaf. The encoding of the components of a constructor is given by

$$\pi(v_1 \dots v_n) = \begin{cases} () & \text{if } n = 0 \\ v_1 & \text{if } n = 1 \\ (\pi(v_1 \dots v_{\lfloor n/2 \rfloor}), \pi(v_{\lfloor n/2 \rfloor+1} \dots v_n)) & \text{if } n > 1 \end{cases} .$$

Of course, we also require the inverse translation, which decodes encoded values. The inverse translation must be applied whenever a generic function produces values whose type depends on the type argument. We leave it to the reader to fill out the details.

Generic definitions can be *implemented* in at least two ways. The semantics sketched above describes an interpretative approach where a generic value repeatedly pattern matches on its type argument. Clearly, this approach is rather inefficient especially since the type argument is statically known at compile time. The interpretative layer can be removed by specializing a generic value with respect to a given type. The process of specialization, which can be seen as a very special instance of partial evaluation, is described at length in [7]—Section 3.1 states the basic idea. Note that the restrictions on the form of generic definitions have been chosen in order to guarantee that specializing a generic definition is always feasible.

2.3 Accessing constructor and label names

The function `encode(t)` can be seen as a very simple printer. To be able to define a *pretty* printer, which produces a human-readable output, we must make the names of the constructors accessible. To this end we provide an additional type pattern of the form `c a` where `c` is a *value* variable of type `Con` and `a` is a *type* variable. The type `Con` is a new primitive type that comprises all constructor names. To manipulate constructor names the following operations can be used (Figure 3 contains a complete list).

```
data Con -- abstract
arity   :: Con -> Int        -- primitive
showCon :: Con -> String     -- primitive
```

Building upon `arity` and `showCon` we can already implement a simple, generic version of Haskell's `showsPrec` function. Recall that `showsPrec(t) d x` takes a precedence level `d` (a value from 0 to 10), a value `x` of type `t` and returns a `String-to-String` function. The following equations define `showsPrec(t)` for all types that do not involve any primitive types.

$$
\begin{array}{lcl}
\mathcal{D}[\![\texttt{data}\ T\ a_1 \ldots a_m\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n]\!] & = & T\ a_1 \ldots a_m = \Sigma(\mathcal{K}[\![c_1]\!] \ldots \mathcal{K}[\![c_n]\!]) \\
\mathcal{D}[\![\texttt{newdata}\ T\ a_1 \ldots a_m\ \texttt{=}\ c]\!] & = & T\ a_1 \ldots a_m = \mathcal{K}[\![c]\!] \\
\mathcal{K}[\![K\ t_1 \ldots t_k]\!] & = & K\ (\Pi(\mathcal{T}[\![t_1]\!] \ldots \mathcal{T}[\![t_k]\!])) \\
\mathcal{K}[\![K\ \{\ell_1 \texttt{::} t_1 \texttt{,} \ldots \texttt{,} \ell_k \texttt{::} t_k\}]\!] & = & K\ (\Pi((\ell_1 \texttt{::} \mathcal{T}[\![t_1]\!]) \ldots (\ell_k \texttt{::} \mathcal{T}[\![t_k]\!]))) \\
\mathcal{T}[\![\texttt{!}\ t]\!] & = & t \\
\mathcal{T}[\![t]\!] & = & t
\end{array}
$$

Figure 1: Translation of datatype declarations and datatype renamings

```
showsPrec(t)                :: Int -> t -> ShowS
showsPrec(a+b) d (Left x)  =  showsPrec(a) d x
showsPrec(a+b) d (Right y) =  showsPrec(b) d y
showsPrec(c a) d x
    | arity c == 0         =  showString (showCon c)
    | otherwise            =  showParen (d >= 10)
                           (  showString (showCon c)
                           .  showChar ' '
                           .  showsPrec(a) 10 x )
showsPrec(a*b) d (x, y) =  showsPrec(a) d x
                        .  showChar ' '
                        .  showsPrec(b) d y
```

The first two equations discard the binary constructors `Left` and `Right`. They are not required since the constructor names are accessible via the type pattern `c a`. If the constructor is nullary, its string representation is emitted. Otherwise, the constructor name is printed followed by a space followed by the representation of its arguments. If the precedence level is 10, the output is additionally parenthesized. The last equation applies if a constructor has more than one component. In this case the components are separated by a space. Note that this definition is a simplified version of `showsPrec(t)` as it neither considers infix operators nor record syntax. A full-blown version appears in Appendix B.

It should be noted that constructor names appear only on the type level; they have no counterpart on the value level as value constructors are encoded using `Left` and `Right`. If a generic definition does not include a case for the type pattern `c a`, then we tacitly assume that `g(c a) = g(a)`.

Haskell allows the programmer to assign labels to the components of a constructor. The names of the labels can be accessed via the type pattern `l::a` where `l` is a *value* variable of type `Label` and `a` is a *type* variable. If a generic definition does not include a case for this pattern, we assume that `g(l::a) = g(a)`. The full-blown version of `showsPrec(t)` uses the type pattern `l::a`.

2.4 Ad-hoc definitions

A generic function solves a problem in a uniform way for all types. Sometimes it is desirable to use a different approach for some datatypes. Consider, for instance, the function `encode` instantiated to lists over some base type. To encode the structure of an n-element list $n+1$ bits are used. For large lists this is clearly wasteful. A more space-efficient scheme stores the length of the list in a header followed by the encodings of the elements. We can specify this compression scheme for lists using a so-called ad-hoc definition.

```
encode([a]) xs =  encodeInt (length xs)
               ++ concatMap (encode(a)) xs
```

Ad-hoc definitions specify exceptions to the general rule and may be given for all predefined and for all user-defined datatypes. As before, the type pattern on the left-hand side must have the form $\mathtt{T}\ \mathtt{a}_1 \ldots \mathtt{a}_k$, ie a type constructor is applied to type variables $\mathtt{a}_1, \ldots, \mathtt{a}_k$, which have to be distinct. Furthermore, `T` must be a type constructor of first-order kind. Recall that the kind system of Haskell specifies the 'type' of a type constructor [14]. The '$*$' kind represents nullary constructors like `Char` or `Int`. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The order of a kind is given by $\mathit{order}(*) = 0$ and $\mathit{order}(\kappa_1 \to \kappa_2) = \max\{1 + \mathit{order}(\kappa_1), \mathit{order}(\kappa_2)\}$. Whether the restriction to first-order kinds is severe remains to be seen.

Ad-hoc definitions can be spread over several modules. Generic definitions are handled very much like class- and instance-declarations. The type signature of a generic definition together with the equations for `1`, '`+`', and '`*`' plays the rôle of a class definition. Ad-hoc definitions are akin to instance declarations. Like instance declarations they cannot be explicitly named on import or export lists [19, Section 5.4]. Rather, they are always implicitly exported.

We have remarked in the introduction that `1`, '`+`', and '`*`' serve as abbreviations for the predefined types `()`, `Either`, and `(,)`. This is, however, not the whole truth. We use two different sets of constructor names to be able to distinguish between generic and ad-hoc equations. For instance, `showsPrec((a,b))` given by

```
showsPrec((a,b)) d (x, y) =  showChar '('
                          .  showsPrec(a) 0 x
                          .  showChar ','
                          .  showsPrec(b) 0 y
                          .  showChar ')'
```

is an ad-hoc definition, which determines the layout of pairs. By contrast, `showsPrec(a*b)` as defined in Section 2.3 is a generic definition specifying the layout of constructor components.

2.5 Mutually recursive definitions

The generic definitions we have seen so far involve only a single generic value. An obvious generalization is to allow for mutually recursive definitions. As before we require that the definitions are inductive on the structure of type expressions. Formally, a mutually recursive declaration group $\mathtt{g}_1, \ldots, \mathtt{g}_n$ is well-formed if there is a partial order on the $\mathtt{g}_i$ such that for every recursive call

$$\mathtt{g}_i(\mathtt{T}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_k) \quad = \quad \ldots\ \mathtt{g}_j(t)\ \ldots$$

either (1) $t \in \{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$ or (2) $\mathtt{g}_i > \mathtt{g}_j$, $t = \mathtt{T}\ \mathtt{b}_1\ \ldots\ \mathtt{b}_k$, and $\{\mathtt{b}_1, \ldots, \mathtt{b}_k\} \subseteq \{\mathtt{a}_1, \ldots, \mathtt{a}_k\}$. Examples for generic definitions that employ this general form can be found in Appendix B.

Generic definitions may only appear in the top-level declaration list of a module. In principle, we could also permit generic definitions in local declaration groups. This extension is, however, hindered by Haskell's lack of scoped type variables. Recall that type signatures are mandatory for generic definitions. The lack of explicit scopes makes it impossible to distinguish between monomorphic and polymorphic uses of type variables. A suitable extension is described in [17].

2.6 Values indexed by first-order type constructors

Functions may not only be indexed by types of kind $*$ but also by type constructors of first-order kind. The archetypical example for such a function is `size(t) :: t a -> Int`, which counts the number of values of type `a` in a given structure of type `t a`. Note that the type parameter of `size` ranges over types of kind $* \to *$. To be able to define `size` generically for all type constructors of this kind we must extend the type language by type abstractions. We agree upon that `\a.t` is syntax for λ-abstractions on the type level. (We use `\a.t` instead of `\a->t` since '`->`' is also used as a type constructor and the type expression `\a->f a->g a` looks rather ugly.) For example, `\a.a` is the identity type and `\a.Char` is the constant type, which maps arbitrary types to `Char`. Using type abstractions the definition of `size` reads

```
size(t)                      :: t a -> Int
size(\a.a)       x           =  1
size(\a.1)       x           =  0
size(\a.Char)    x           =  0
size(\a.Int)     x           =  0
size(\a.f a+g a) (Left x)    =  size(f) x
size(\a.f a+g a) (Right y)   =  size(g) y
size(\a.f a*g a) (x, y)      =  size(f) x
                             +  size(g) y
```

To understand the equations it is helpful to replace the formal type parameter `t` in the type signature by the respective type arguments. For the first equation we obtain the type `a->Int`. Clearly, a structure of type `a` contains one element of type `a`. Substituting `\a.1` for `t` yields `1->Int`, ie we have no elements of type `a`. For sums the substitution yields `f a+g a->Int`. To determine the size of an element of type `f a+g a` we must either calculate the size of a structure of type `f a` or that of a structure of type `g a`. Finally, the size of a structure of type `f a*g a` is given by the sum of the size of the two components.

The definition of `size` employs two sorts of type patterns: the identity type `\a.a`, which may be abbreviated by `Id`, and patterns of the form `\a.T` $(\mathtt{f}_1\ \mathtt{a})\ \ldots\ (\mathtt{f}_k\ \mathtt{a})$ where the $\mathtt{f}_i$ are type variables of kind $* \to *$. To be exhaustive a generic definition must include cases for `Id`, `1`, '`+`', and '`*`' and additionally for all primitive types. Now, reconsider the definition of `size` and note that the cases for constant types of the form `\a.T` are all alike. We allow the programmer to combine these cases by using the type pattern `\a.u`, which may be abbreviated to `K u`. Using this shortcut the definition of `size` can be rewritten to

```
size(t)                      :: t a -> Int
size(Id)  x                  =  1
size(K u) x                  =  0
size(\a.f a+g a) (Left x)    =  size(f) x
size(\a.f a+g a) (Right y)   =  size(g) y
size(\a.f a*g a) (x, y)      =  size(f) x
                             +  size(g) y
```

Applying a generic value works as before: the value is called with a closed type expression. The type may also involve type abstractions. For example, `size(\a->[[a]])` counts the number of elements in a list of lists.

Let us stress that type arguments cannot be unambiguously inferred by the compiler—this is the principle reason why they are mandatory. Consider, for instance, the call `size xs` where `xs` has type `[[Int]]`. To determine the type argument of `size` we must solve the equation `t u = [[Int]]`. The higher-order unification of `t u` and `[[Int]]` yields, however, three different solutions: `t` = `\a.[a]` and `u` = `[Int]`, `t` = `\a->[[a]]` and `u` = `Int`, or `t` = `\a->[[Int]]` and `u` arbitrary. To prevent ambiguity type arguments must never be omitted. Alternatively, one could use Haskell's kinded first-order unification [14], which produces unique solutions (the first one in the example above). Some practical experience is needed to decide whether this is a viable alternative.

A value may be indexed by an arbitrary first-order type constructor. In the general case, the type patterns are either projection patterns or constructor patterns. If the type parameter has n arguments, we have n different projection patterns: $\backslash\mathtt{a}_1 \ldots \mathtt{a}_n.\mathtt{a}_1$, $\ldots$, $\backslash\mathtt{a}_1 \ldots \mathtt{a}_n.\mathtt{a}_n$. Constructor patterns have the general form $\backslash\mathtt{a}_1 \ldots \mathtt{a}_n.\mathtt{T}\ (\mathtt{f}_1\ \mathtt{a}_1 \ldots \mathtt{a}_n)\ \ldots\ (\mathtt{f}_k\ \mathtt{a}_1 \ldots \mathtt{a}_n)$. Note that ad-hoc definitions employ constructor patterns, as well. For instance, an ad-hoc definition for the list type has the form $\mathtt{g}(\backslash\mathtt{a}_1 \ldots \mathtt{a}_n.[\mathtt{f}\ \mathtt{a}_1 \ldots \mathtt{a}_n]) = \ldots \mathtt{g}(\mathtt{f}) \ldots$.

3 Blending generic definitions with type classes

Type classes and generic definitions are closely related concepts. This section is concerned with the integration of the two features. In particular, we show how to specify derivable class methods by generic definitions. Beforehand, let us briefly comment on the differences between type classes and generic definitions. A type class corresponds roughly to the type signature of a generic definition—or rather, to a collection of type signatures. Instance declarations are related to ad-hoc definitions. Thus, class and instance declarations can be mimicked by generic definitions. Since generic definitions furthermore allow instances to be defined in a uniform way, they are, in fact, more general than type classes. On the negative side, the user must always supply explicit type arguments. We have seen in Section 2.6 that it is not possible to infer the type arguments of generic calls. By contrast, class methods can, in general, be called without giving explicit type information.

Now, integrating generic definitions with type classes we (hope to) get the best of both worlds. To this end we allow default class methods to be specified by generic definitions. That way a class method can be defined generically for all instances making it possible to *derive* instances for newly defined datatypes. Take, for example, Haskell's `Eq` class, which is one of the few derivable classes. Here is a generic definition of `Eq`.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

```
(==)(1)    ()         ()         =  True
(==)(a+b)  (Left  v)  (Left  x)  =  (==)(a) v x
(==)(a+b)  (Left  v)  (Right y)  =  False
(==)(a+b)  (Right w)  (Left  x)  =  False
(==)(a+b)  (Right w)  (Right y)  =  (==)(b) w y
(==)(a*b)  (v, w)     (x, y)     =  (==)(a) v x
                                 && (==)(b) w y
x /= y       =  not (x==y)
```

Given this declaration instances may be derived for arbitrary types of kind $*$ and for arbitrary type constructors of first-order kind—only subsidiary primitive types must be instances of `Eq`. For type constructors of first-order kind the derived instance declaration has the form `(Eq` a_1`, ..., Eq` a_k`) => Eq (T` $a_1 \ldots a_k$`)`. It is important to note that instances can be derived no matter of the internal structure of the types. For example, the following definition

```
data Matrix a = Matrix (Twice [] a) deriving (Eq)
```

which uses the second-order type constructor `Twice` defined in the introduction, is perfectly legal. Currently, this definition is rejected.

Generic default methods can also be specified for classes that abstract over first-order type constructors. Haskell's `Functor` class serves as an excellent example. The `Functor` class consists of the so-called mapping function `map(t)`, which applies a given function to each element of type `a` in a given structure of type `t a`.[1] It is well-known that `map(t)` can be defined for all type constructors of kind $* \to *$ that do not involve the function space. Alas, currently the user must program instances of `Functor` by hand (which is for the most part tedious but sometimes quite involving). Again, generic default methods save the day.

```
class Functor t where
  map :: (a -> b) -> (t a -> t b)

  map(Id)         h x         =  h x
  map(K u)        h x         =  x
  map(\a.f a+g a) h (Left x)  =  Left (map(f) h x)
  map(\a.f a+g a) h (Right y)=  Right(map(g) h y)
  map(\a.f a*g a) h (x, y)    =  (map(f) h x,
                                  map(g) h y)
```

Now, instances of `Functor` can be derived for arbitrary type constructors of kind $* \to *$ (using `deriving`). Note that Appendix B defines a slightly more general version that also takes the function space into account.

3.1 Limits of type classes

We have noted in the introduction that instances cannot be derived for all datatypes in Haskell 98. In this section we take a brief look at the reasons for this failure. Consider the following datatype declarations taken from [6] (note that `MapF` is isomorphic to `Twice`).

```
data MapF m v =  MapF (m (m v))
data MapS m v =  MapS v (MapS (MapF m) v)
                        (m (MapS (MapF m) v))
```

Both `MapF` and `MapS` are second-order type constructors of kind $(* \to *) \to (* \to *)$. The attempt to derive an instance of a predefined class, say, `Eq` fails in both cases. Recall that instance declarations for `Eq` have essentially the following form (the context may additionally refer to other classes).

`instance (Eq` a_1`, ..., Eq` a_k`) => Eq (T` $a_1 \ldots a_k$`)`

Clearly, this form is too limited to handle `MapF` or `MapS`. In [20] an extension of the class system is described, which allows for more general instance contexts. Using this extension, which has been realized in GHC [18] and in Hugs 98 [15], the following `Eq` instance can be derived for `MapF` (do you guess how `(==)` is implemented?).

```
instance (Eq (m (m v))) => Eq (MapF m v)
```

However, if we try to derive `Eq` for `MapS`, both compilers loop! Even if the generalized form of instance contexts is used, the type constraints cannot be finitely represented. This example demonstrates the limits of Haskell's class system.

On the other hand, generic definitions *can* be specialized for higher-order type constructors such as `MapF` and `MapS`. Let us sketch the main idea in the rest of this section—the technique is described at length in [7]. Let `EqD a` be the dictionary type corresponding to the `Eq` class. First of all, note that an instance declaration of the form

```
instance (Eq a) => Eq (T a)
```

defines a function of type `∀a.EqD a -> EqD (T a)`, ie the dictionary for `a` is mapped to the dictionary for `T a`. Here `T` is a type constructor of kind $* \to *$. Consequently, an instance for a type constructor `H` of kind $(* \to *) \to (* \to *)$ should be a function that maps $* \to *$ instances to $* \to *$ instances.

```
∀f.(∀a.(EqD a -> EqD (f a)))
   -> (∀b.(EqD b -> EqD (H f a)))
```

This scheme can be easily generalized to type constructors of even higher-order kinds. So we have the unfortunate situation that `Eq` instances can, in principle, be derived for arbitrary type constructors but these instances cannot be represented as Haskell instance declarations. This explains, in particular, why `deriving` is restricted to type constructors of kind $*$ or of first-order kind though the type may internally depend on types of higher-order kinds.

4 Examples

This section presents further examples for generic functions. The first example, `decode(t)`, is essentially the inverse of `encode(t)`: it takes a bit string, decodes a prefix of that bit string, and returns the decoded value coupled with the unused suffix. The two functions are related by `decode(t) (encode(t) x ++ bs) = (x, bs)`.

```
decode(t)             :: [Bit] -> (t, [Bit])
decode(1) bs          =  ((), bs)
decode(Int) bs        =  decodeInt bs
decode(Char) bs       =  decodeChar bs
decode(a+b) []        =  error "decode"
decode(a+b) (O : bs)  =  appl Left  (decode(a) bs)
decode(a+b) (I : bs)  =  appl Right (decode(b) bs)
decode(a*b) bs        =  let (x, cs) = decode(a) bs
                             (y, ds) = decode(b) cs
                         in  ((x, y), ds)
appl f (x, y)         =  (f x, y)
```

[1] In Haskell 98 `map` is called `fmap`.

The ad-hoc definition for lists, `decode([a])`, is left as an instructive exercise to the reader.

The Standard Prelude contains a number of list processing functions, which can be generalized to arbitrary type constructors: `and`, `or`, `all`, `any`, `sum`, `product` etc. They are all instances of a more general concept termed reduction or crush [16]. A crush is a function of type `t a -> a` that collapses a structure of values of type `a` into a single value of type `a`. To define a crush we require two ingredients: a value `e :: a` and a binary operation `h :: a->a->a`. Usually but not necessarily `e` is the neutral element of `h`.

```
crush(t) :: a -> (a -> a -> a) -> (t a -> a)
crush(Id)        e h x          =   x
crush(K u)       e h x          =   e
crush(\a.f a+g a)e h (Left x) =   crush(f) e h x
crush(\a.f a+g a)e h (Right y)=   crush(g) e h y
crush(\a.f a*g a)e h (x, y)   =   crush(f) e h x
                                `h` crush(g) e h y
```

Using `crush(t)` we can define, for instance, generic versions of `and` and `sum`.

```
and(t)   :: t Bool -> Bool
and(t)   =  crush(t) True (&&)
sum(t)   :: (Num n) => t n -> n
sum(t)   =  crush(t) 0 (+)
```

Further examples can be found in Appendix B or in [16, 10].

The function `sum(t)` shows that type signatures of generic values may also involve class constraints. The following definition, which implements a monadic mapping function [3], falls back on the `Monad` class. A monadic map applies a function of type `a->m b` to all elements of type `a` in a given structure of type `t a` and then threads the monad `m` from left to right through the resulting structure.

```
mapM(t) :: Monad m => (a->m b) -> (t a->m (t b))
mapM(Id)  h x = h x
mapM(K u) h x = return x
mapM(\a.f a+g a) h (Left x)
    = do { v <- mapM(f) h x; return(Left v) }
mapM(\a.f a+g a) h (Right y)
    = do { w <- mapM(g) h y; return(Right w) }
mapM(\a.f a*g a) h (x, y)
    = do { v <- mapM(f) h x; w <- mapM(g) h y;
           return (v, w) }
```

Note that `mapM` generalizes the function of the same name defined in the Standard Prelude (the Prelude function is specialized to lists). An important special case of `mapM` is

```
thread(t)    :: (Monad m) => t (m a) -> m (t a)
thread(t)    =  mapM(t) id
```

The function `thread(t)` essentially commutes the type constructors `t` and `m`. It generalizes the Prelude function `sequence`, which operates on lists.

5 Related work

The concept of generic functional programming trades under a variety of names: F. Ruehr refers to this concept as *structural polymorphism* [22, 21], T. Sheard calls generic functions *type parametric* [23], C.B. Jay and J.R.B. Cocket use the term *shape polymorphism* [12], R. Harper and G. Morrisett [5] coined the phrase *intensional polymorphism*, and J. Jeuring invented the word *polytypism* [13].

The mainstream of generic programming is based on the initial algebra semantics of datatypes [4] and typically requires a basic knowledge of category theory. The categorical programming language Charity [1] automatically provides `map` and catamorphisms for each user-defined datatype. Functorial ML [11] has a similar functionality but a different background. It is based on the theory of *shape polymorphism*, in which values are separated into shape and contents. The polytypic programming language extension PolyP [9] offers a special construct for defining generic functions. The generic definitions are similar to the ones given in this paper (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion (see [8] for a detailed comparison). All the approaches with the notable exception of [22] are restricted to first-order kinded, regular datatypes (or even subsets of this class), that is they cover only a small part of Haskell's type system.

The theoretical background and a possible implementation of the generic programming extension presented here are described in a companion paper [7], which contains supplementary pointers to relevant work. We show, in particular, how to specialize generic definitions for given instances of datatypes. The specialization employs a generalization of the dictionary passing translation and does not require the passing of types or representations of types at run-time. This is in contrast to the work on *intensional polymorphism* [5, 2] where a **typecase** is used for defining type-dependent operations.

6 Conclusion

This paper proposes a generic programming extension for Haskell. We feel that the extension fits quite smoothly into the language. In particular, it generalizes the `deriving` construct and makes derivable class methods explicit. The examples given in the paper furthermore demonstrate the usefulness of such an extension.

Acknowledgements

I would like to thank Phil Wadler and four anonymous referees for many helpful comments.

A Changes to the syntax

Figure 2 lists the necessary changes to the syntax. The syntax of type expressions has been extended to include type abstractions, '+', and '*'. Optional type arguments have been added to type signatures, to the left-hand sides of function definitions, and to value expressions.

B Changes to the Standard Prelude and to the Standard Libraries

The Standard Prelude is extended by a module called `Generics`, see Figure 3, which defines `Con`, `Label`, and a few generic values. Figures 4, 5, 6, and 7 contain the modifications to the Standard Prelude. Note that all derivable classes (`Eq`, `Ord`, `Enum`, `Bounded`, `Show`) with the exception of `Read` are precisely defined through generic default methods. The definition of `Read` is omitted for reasons of space. Finally, Figure 8 lists the changes to the Standard Library `Ix`.

gendecl	$\rightarrow$	*gvars* `::` [*context* `=>`] *type*	(type signature)
	$\mid$	$\ldots$	
gvars	$\rightarrow$	gvar_1 `,` $\ldots$ `,` gvar_n	$(n \geqslant 1)$
gvar	$\rightarrow$	*var*	
	$\mid$	*var tyvar*	(type-indexed value)
type	$\rightarrow$	type^0	
type^0	$\rightarrow$	type^1 [`->` type^0]	(function type)
type^1	$\rightarrow$	type^2 [`+` type^1]	(binary sums)
type^2	$\rightarrow$	type^3 [`*` type^2]	(binary products)
type^3	$\rightarrow$	`\` tyvar_1 $\ldots$ tyvar_n `.` *type*	(type abstraction, $n \geqslant 1$)
	$\mid$	*var atype*	(constructor type pattern)
	$\mid$	*var* `::` *btype*	(label type pattern)
	$\mid$	*btype*	(type application)
gtycon	$\rightarrow$	*qtycon*	
	$\mid$	`()`	(unit type)
	$\mid$	`1`	(unit type)
	$\mid$	`[]`	(list constructor)
	$\mid$	`(->)`	(function constructor)
	$\mid$	`(+)`	(binary sum constructor)
	$\mid$	`(*)`	(binary product constructor)
	$\mid$	`(,{,})`	(tupling constructors)
funlhs	$\rightarrow$	*var atype* { *apat* }	(type-indexed value)
	$\mid$	$\ldots$	
fexp	$\rightarrow$	*qvar atype* { *aexp* }	(application of a type-indexed value)
	$\mid$	$\ldots$	

Figure 2: Changes to the syntax

```
module Generics(
    Con, arity, showCon, hasLabels, Fixity(Nonfix, Infix, Infixl, Infixr, prec), fixity,
    precedence, Label, showLabel, card, crush, gsum, gproduct, size, gand, gor, gall, gany,
    gelem, gnotElem, listify ) where

-- abstract types and primitives

data Con  -- abstract
arity     :: Con -> Int     -- primitive
showCon   :: Con -> String  -- primitive
hasLabels :: Con -> Bool    -- primitive
instance Eq Con ...
instance Show Con where show = showCon

data Fixity = Nonfix | Infix {prec :: Int} | Infixl {prec :: Int} | Infixr {prec :: Int}
              deriving (Eq, Ord, Read, Show)
fixity       :: Con -> Fixity  -- primitive
precedence   :: Con -> Int
precedence c =  prec (fixity c)

data Label  -- abstract
showLabel :: Label -> String  -- primitive
instance Eq Label ...
instance Show Label where show = showLabel

-- generic definitions

card(t)    :: int     -- NB: card(t) is undefined for infinite types
card(1)    =  1
card(Char) =  ...     -- system dependent
card(Int)  =  ...     -- system dependent
card(a+b)  =  card(a) + card(b)
card(a*b)  =  card(a) * card(b)
card(a->b) =  card(b) ^ card(a)

crush(t)                         :: a -> (a -> a -> a) -> (t a -> a)
crush(Id)         e op x         =  x
crush(K u)        e op x         =  e
crush(\a.f a+g a) e op (Left x)  =  crush(f) e op x
crush(\a.f a+g a) e op (Right y) =  crush(g) e op y
crush(\a.f a*g a) e op (x, y)    =  crush(f) e op x `op` crush(g) e op y

gsum(t), gproduct(t)     :: (Num n) => t n -> n
gsum(t)                  =  crush(t) 0 (+)
gproduct(t)              =  crush(t) 1 (*)
size(t)                  :: (Num n) => t a -> n
size(t)                  =  gsum(t) . map(t) (const 1)
gand(t), gor(t)          :: t Bool -> Bool
gand(t)                  =  crush(t) True (&&)
gor(t)                   =  crush(t) False (||)
gall(t), gany(t)         :: (a -> Bool) -> (t a -> Bool)
gall(t) p                =  gand(t) . map(t) p
gany(t) p                =  gor(t)  . map(t) p
gelem(t), gnotElem(t)    :: (Eq a) => a -> t a -> Bool
gelem(t) x               =  gany(t) (== x)
gnotElem(t) x            =  gall(t) (/= x)
listify(t)               :: t a -> [a]
listify(t) x             =  crush(t) id (.) (map(t) (:) x) []
```

Figure 3: Module Generics

```
module Prelude
...

import Generics

-- Equality and Ordered classes

class  Eq t  where
  (==), (/=) :: t -> t -> Bool

  (==) (t)                          :: t -> t -> Bool
  (==) (1)   ()         ()          =  True
  (==) (a+b) (Left  x1) (Left  y1) =  (==) (a) x1 y1
  (==) (a+b) (Left  x1) (Right y2) =  False
  (==) (a+b) (Right x2) (Left  y1) =  False
  (==) (a+b) (Right x2) (Right y2) =  (==) (b) x2 y2
  (==) (a*b) (x1, x2)   (y1, y2)   =  (==) (a) x1 y1 && (==) (b) x2 y2

  x /= y     =  not (x==y)

class  (Eq t) => Ord t  where
  compare              :: t -> t -> Ordering
  (<), (<=), (>=), (>) :: t -> t -> Bool
  max, min             :: t -> t -> t

  compare(1)   ()         ()         =  EQ
  compare(a+b) (Left  x1) (Left  y1) =  compare(a) x1 y1
  compare(a+b) (Left  x1) (Right y2) =  LT
  compare(a+b) (Right x2) (Left  y1) =  GT
  compare(a+b) (Right x2) (Right y2) =  compare(b) x2 y2
  compare(a*b) (x1, x2)   (y1, y2)   =  compare(a) x1 y1 `lex` compare(b) x2 y2
    where lex LT rel =  LT
          lex EQ rel =  rel
          lex GT rel =  GT
  ...

-- Enumeration and Bounded classes

class  Enum t  where
  succ, pred      :: t -> t
  toEnum          :: Int -> t
  fromEnum        :: t -> Int
  enumFromTo      :: a -> a -> [a]        -- [n..m]
  ...

  toEnum(1) n
    | n == 0      =  ()
    | otherwise   =  error "toEnum: Out of range"
  toEnum(a+b) n
    | n < card(a) =  Left  (toEnum(a) n)
    | otherwise   =  Right (toEnum(b) (n - card(a)))
  toEnum(a*b) n   =  (toEnum(a) r, toEnum(b) q)
    where (q, r)  =  divMod n (card(a))

  fromEnum(1) ()          =  0
  fromEnum(a+b) (Left x)  =  fromEnum(a) x
  fromEnum(a+b) (Right y) =  card(a) + fromEnum(b) y
  fromEnum(a*b) (x, y)    =  fromEnum(a) x + card(a) * fromEnum(b) y

  enumFromTo(a) x y =  map (toEnum(a)) [fromEnum(a) x .. fromEnum(a) y]
  ...
```

Figure 4: Changes to `Prelude`

```
class  Bounded t  where
  minBound      :: t
  maxBound      :: t

  minBound(1)   =  ()
  minBound(a+b) =  Left (minBound(a))
  minBound(a*b) =  (minBound(a), minBound(b))

  maxBound(1)   =  ()
  maxBound(a+b) =  Right (maxBound(b))
  maxBound(a*b) =  (maxBound(a), maxBound(b))

-- Monadic classes

class Functor t where
  map   :: (a -> b) -> (t a -> t b)
  comap :: (a -> b) -> (t b -> t a)

  map(Id)          h x         =  h x
  map(K u)         h x         =  x
  map(\a.f a+g a)  h (Left x)  =  Left  (map(f) h x)
  map(\a.f a+g a)  h (Right y) =  Right (map(g) h y)
  map(\a.f a*g a)  h (x, y)    =  (map(f) h x, map(g) h y)
  map(\a.f a->g a) h phi       =  map(g) h . phi . comap(f) h

  comap(K u)         h x         =  x
  comap(\a.f a+g a)  h (Left x)  =  Left  (comap(f) h x)
  comap(\a.f a+g a)  h (Right y) =  Right (comap(g) h y)
  comap(\a.f a*g a)  h (x, y)    =  (comap(f) h x, comap(g) h y)
  comap(\a.f a->g a) h phi       =  comap(g) h . phi . map(f) h
...

data Maybe a = Nothing | Just a deriving (Eq, Ord, Read, Show, Functor)
...
```

Figure 5: Changes to `Prelude` (continued)

```
module PreludeList
...
import Generics

...
sum, product     :: (Num a) => [a] -> a
sum              =  gsum([])
product          =  gproduct([])
length           :: [a] -> Int
length           =  size([])
and, or          :: [Bool] -> Bool
and              =  gand([])
or               =  gor([])
any, all         :: (a -> Bool) -> [a] -> Bool
any              =  gany([])
all              =  gall([])
elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem             =  gelem([])
notElem          =  gnotElem([])
...
```

Figure 6: Changes to `PreludeList`

```
module PreludeText
...
import Generics

class  Show t  where
  showsPrec        :: Int -> t -> ShowS
  show             :: t -> String
  showList         :: [t] -> ShowS
  ...
  showsPrec(a+b)  d (Left x)  =  showsPrec(a) d x
  showsPrec(a+b)  d (Right y) =  showsPrec(b) d y
  showsPrec(a*b)  d (x, y)    =  showsPrec(a) d x . showChar ' ' . showsPrec(b) d y
  showsPrec(a->b) d phi       =  showString "<function>"
  showsPrec(c a)  d x
    | hasLabels c             =  shows c . showChar '{' . showsRecord(a) x . showChar '}'
    | arity c == 0            =  shows c
    | fixity c /= Nonfix      =  showsInfix(a) d c x
    | otherwise               =  showParen (d >= 10) (shows c . showChar ' ' . showsPrec(a) 10 x)

  showsRecord(1)    ()        =  showString ""
  showsRecord(a+b)  x         =  showsPrec(a+b) 0 x
  showsRecord(a*b)  (x, y)    =  showsRecord(a) x . showChar ',' . showsRecord(b) y
  showsRecord(a->b) phi       =  showsPrec(a->b) 0 phi
  showsRecord(c a)  x         =  showsPrec(c a) 0 x
  showsRecord(l::a) x         =  shows l . showChar '=' . showsPrec(a) 0 x

  showsInfix(a*b) d c (x, y)  =  showParen (d > precedence c) (showi (fixity c))
    where op                  =  showString (" " ++ showCon c ++ " ")
          showi (Infix  p)    =  showsPrec(a) (p+1) x . op . showsPrec(b) (p+1) y
          showi (Infixl p)    =  showsPrec(a) p     x . op . showsPrec(b) (p+1) y
          showi (Infixr p)    =  showsPrec(a) (p+1) x . op . showsPrec(b) p     y

class  Read t  --- analogous
```

Figure 7: Changes to `PreludeText`

```
module Ix
...
class  (Ord t) => Ix t  where
  range               :: (t,t) -> [t]
  index               :: (t,t) -> t -> Int
  inRange             :: (t,t) -> t -> Bool
  rangeSize           :: (t,t) -> Int

  range(1)   (l,h)           =  [()]
  range(a+b) (l,h)           =  enumFromTo(a+b) l h
  range(a*b) ((l,l'),(h,h')) =  [(i,i') | i <- range (l,h), i' <- range (l',h')]

  index(1)     (l,h) i =  0
  index(a+b) b@(l,h) i
    | inRange(a+b) b i =  fromEnum(a+b) i - fromEnum(a+b) l
    | otherwise        =  error "Ix.index: Index out of range."
  index(a*b) ((l,l'),(h,h')) (i,i')
                       =  index(a) (l,h) i * rangeSize(b) (l',h') + index(b) (l',h') i'

  inRange(1)   (l,h)           i      =  True
  inRange(a+b) (l,h)           i      =  compare(a+b) l i /= GT && compare(a+b) i h /= GT
  inRange(a*b) ((l,l'),(h,h')) (i,i') =  inRange(a) (l,h) i && inRange(b) (l',h') i'

  rangeSize(a) b@(l,h) | null (range(a) b) = 0
                       | otherwise         = index(a) b h + 1
```

Figure 8: Changes to `Ix`

References

[1] Robin Cockett and Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.

[2] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices*, 34(1):301–312, 1999.

[3] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report Memoranda Informatica 94-28, University of Twente, June 1994.

[4] T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.

[5] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, San Francisco, California*, pages 130–141. ACM-Press, 1995.

[6] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 1999. Accepted for publication.

[7] Ralf Hinze. A new approach to generic functional programming. Technical Report IAI-TR-99-9, Institut für Informatik III, Universität Bonn, July 1999.

[8] Ralf Hinze. Polytypic programming with ease (extended abstract). In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Lecture Notes in Computer Science. Springer-Verlag, November 1999. To appear.

[9] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pages 470–482. ACM-Press, January 1997.

[10] Patrik Jansson and Johan Jeuring. PolyLib—A library of polytypic functions. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden.* Department of Computing Science, Chalmers University of Technology and Göteborg University, June 1998.

[11] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[12] C.B. Jay and J.R.B. Cocket. Shapely types and shape polymorphism. In D. Sanella, editor, *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316, Berlin, 11–13 April 1994. Springer-Verlag.

[13] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[14] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[15] M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.

[16] Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.

[17] Erik Meijer and Koen Claessen. The design and implementation of Mondrian. In *Proceedings of the Haskell Workshop*, 1997.

[18] Simon Peyton Jones. Explicit quantification in Haskell, 1998. Available from `http://research.microsoft.com/Users/simonpj/Haskell/quantification.html`.

[19] Simon Peyton Jones and John Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language*, February 1999.

[20] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: Exploring the design space. In *Proceedings of the Haskell Workshop*, 1997.

[21] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.* Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[22] Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan, 1992.

[23] Tim Sheard. Type parametric programming. Technical report, Oregon Graduate Institute of Science and Technology, Portland, OR, USA, 1993.