

Libraries for Generic Programming in Haskell

Johan Jeuring, Sean Leather, José Pedro Magalhães, and
Alexey Rodriguez Yakushev

Utrecht University , The Netherlands

Abstract. These lecture notes introduce libraries for datatype-generic programming in Haskell. We introduce three characteristic generic programming libraries: lightweight implementation of generics and dynamics, extensible and modular generics for the masses, and scrap your boilerplate. We show how to use them to use and write generic programs. In the case studies for the different libraries we introduce generic components of a medium-sized application which assists a student in solving mathematical exercises.

1 Introduction

One of the most important activities in software development is *structuring data*. Many programming methods and software development tools center around creating a datatype (or XML schema, UML model, class, grammar, etc.). Once the structure of the data has been designed, a software developer adds *functionality* to the datatypes. There is always some functionality that is specific for a datatype, and part of the reason why the datatype has been designed in the first place. Other functionality is similar or even the same on many datatypes. Examples of such functionality are:

- in a large datatype, looking for occurrences of a particular constructor (e.g., for representing variables) for which we want to do something, ignoring the rest of the value;
- functions that depend only on the *structure* of the datatype, such as the equality function;
- adapting the code after datatypes have changed or evolved.

Generic programming addresses these high-level programming patterns. We also use the term datatype-generic programming to distinguish the field from Java generics, Ada generic packages, generic programming in C++ STL, etc. Using generic programming, we can easily implement traversals in which a user is only interested in a small part of a possibly large value, functions which are naturally defined by induction on the structure of datatypes, and functions that automatically adapt to a changing datatype. Larger examples of generic programming include XML tools, testing frameworks, debuggers, and data conversion tools.

Until recently, an instance of a datatype-generic program on a particular datatype was obtained by implementing the instance by hand, a boring and

error-prone task, which reduces programmers' productivity. Some programming languages provide standard implementations of basic datatype-generic programs such as equality of two values and printing a value. In this case, the programs are integrated into the language, and cannot be extended or adapted. So, how can we define datatype-generic programs ourselves?

More than a decade ago the first programming languages appeared that supported the definition of datatype-generic programs. Using these programming languages it is possible to define a generic program which can then be used on a particular datatype without further work. Although these languages allow us to define our own generic programs, they have never grown out of the research prototype phase, and most cannot be used anymore.

The rich type system of Haskell allows us to write a number of datatype-generic programs in the language itself. The power of classes, constructor classes, functional dependencies, generalized algebraic datatypes, and other advanced language constructs of Haskell is impressive, and since 2001 we have seen at least 10 proposals for generic programming libraries in Haskell using one or more of these advanced constructs. Using a library instead of a separate programming language for generic programming has many advantages. The main advantages are that a user does not need a separate compiler for generic programs and that generic programs can be used out of the box. Furthermore, a library is much easier to ship, support, and maintain than a programming language, which makes the risk of using generic programs smaller. The loss of expressiveness compared with a generic programming language such as Generic Haskell is limited.

These lecture notes introduce generic programming in Haskell using libraries. We introduce several characteristic generic programming libraries, and we show how to use them to use and write generic programs. Furthermore, in the case studies for the different libraries we introduce generic components of a medium-sized application which assists a student in solving mathematical exercises. We have included several exercises in these lecture notes. The answers to these exercises can be found in the technical report accompanying these notes [Jeuring et al., 2008].

These notes are organised as follows. Section 2 puts generic programming into context. It introduces a number of variations on the theme of generics as well as demonstrates how each may be used in Haskell. Section 3 reveals our focus on datatype-generic programming by discussing the world of datatypes supported by Haskell and common extensions. In Section 4, we introduce libraries for generic programming and briefly discuss the criteria we used to select the libraries covered in the following three sections. Section 5 starts the discussion of libraries with Lightweight Implementation of Generics and Dynamics (LIGD); however, we leave out the dynamics in order to focus on generics in this review. Section 6 continues with a look at Extensible and Modular Generics for the Masses (EMGM), a library using the same view as LIGD but implemented with a different mechanism. Section 7 examines Scrap Your Boilerplate (SYB), a library implemented with combinators and quite different from LIGD

and EMGM. After describing each library individually, we provide an abridged comparison of them in Section 8. In Section 9, we take a detour from the discussion of established libraries to explore how type-indexed datatypes can be implemented using type families. Finally, we conclude in Section 10 with some suggested reading and some thoughts about the future of generic programming libraries.

2 Generic programming

Generic programming has developed as a technique for increasing the amount and scale of reuse in code while still preserving type safety. The term “generic” is highly overloaded in computer science; however, broadly speaking, most uses involve some sort of parametrisation. A generic program abstracts over the differences in separate but similar programs. In order to arrive at specific programs, one instantiates the parameter in various ways. It is the type of the parameter that distinguishes each variant of generic programming.

There are many variations of generic programming. Gibbons [Gibbons, 2007] lists seven broad categories which we revisit in this section. We focus on how each category has been or can be implemented within Haskell.

Each of the following sections is titled according to the type of the parameter of the generic abstraction. In each, we provide a description of that particular form of generics along with an example of how to apply the technique.

2.1 Value

The most basic form of generic programming is to parametrise a computation by values. The idea goes by various names in programming languages (procedure, subroutine, function), and it is a fundamental element in mathematics. While parametrisation by value is not often considered under the definition of “generic,” it is perfectly reasonable to model other forms of genericity as functions. In this case, the function $g(x)$ represents a generic component g that is parametrised by an entity x . Instantiation of the generic component is then analogous to application of a function.

In Haskell, functions come naturally. For example, here is function that takes two Boolean values as arguments and determines their basic equality.

$$\begin{aligned} eq_{Bool} &:: Bool \rightarrow Bool \rightarrow Bool \\ eq_{Bool} \ x \ y &= (not \ x \wedge not \ y) \vee (x \wedge y) \end{aligned}$$

2.2 Type

Commonly known as *polymorphism*¹, genericity by type refers to both type abstractions (types parametrised by other types) and polymorphic functions (functions with polymorphic types).

¹ Specifically, Haskell supports *parametric polymorphism*. There are other flavors of polymorphism such as subtype polymorphism that we elide.

Haskell has excellent support for polymorphic datatypes and functions. The canonical examples are the `List` datatype and the `length` function. Here is the datatype for a list of Boolean values.

```
data ListB = NilB | ConsB Bool ListB
```

We then offer a simple function for calculating the length of the list.

```
length :: ListB → Int
length NilB           = 0
length (ConsB x xs) = 1 + length xs
```

By changing the type `Bool` in the definition of `ListB` to another type, we get lists of values of this other type. Notice also that `length` never makes use of the actual elements of the list, rather it only counts them. We can therefore abstract over the type of the element and preserve the structure by redefining the `ListB` datatype and the `length` function:

```
data List a = Nil | Cons a (List a)

length :: List a → Int
length Nil           = 0
length (Cons x xs) = 1 + length xs
```

`List a` is a datatype parametrised by another type, and `length` is now a polymorphic function that can be applied to a value of type `List a` for any `a`. Note how the implementation of `length` remains the same; only the type has changed. We simply apply `length` to some list to arrive at an instance for a particular concrete type².

```
length (Cons True (Cons False (Cons True Nil))) ~ 3
```

The datatype `ListB` is a *monomorphic* type: it has only one type. Polymorphic datatypes such as `List` are parametrised and thus may be instantiated to many different types, such as `List Bool` and `List Int`. Haskell supports a wide range of datatypes, enough in fact to make it necessary for a separate section to cover. We reserve Section 3 for this purpose.

2.3 Function

If a function is a first-class citizen in a programming language, parametrisation of one function by another function is exactly the same as parametrisation by value. However, we explicitly mention this category because it enables abstraction over control flow. The full power of function parameters can be seen in the *higher-order functions* of languages such as Haskell and ML.

² We use the notation $a \rightsquigarrow b$ to mean that, in GHCi, expression a evaluates to b .

Suppose we have a list of Boolean values and we want to determine both the logical conjunction and logical disjunction of all values. We can define these functions as follows.

```
and :: List Bool → Bool
and Nil          = True
and (Cons p ps) = p ∧ and ps
```

```
or :: List Bool → Bool
or Nil          = False
or (Cons p ps) = p ∨ or ps
```

These two functions exhibit the same recursive pattern. To abstract from this pattern, we look at the differences between *and* and *or*, and abstract over those components.

```
foldr :: (a → b → b) → b → List a → b
foldr f n Nil          = n
foldr f n (Cons x xs) = f x (foldr f n xs)
```

The pattern that is extracted is known in the Haskell standard library as *foldr* (“fold from the right”). It captures the essence of the recursion in the functions *and* and *or*, and accepts parameters for the binary logical operator and the value for the *Nil* case. Using *foldr*, we redefine *and* and *or* in simpler terms.

```
and = foldr (∧) True
or  = foldr (∨) False
```

2.4 Interface

A generic program may abstract over a given set of requirements. In this case, a specific program can only be instantiated by parameters that conform to these requirements, and the generic program remains unaware of any unspecified aspects. Gibbons calls the set of required operations the “structure” of the parameter; however, we think this may easily be confused with generics by the shape of a datatype (discussed in Section 2.7). Instead, we define *interface* as the set of requirements needed for instantiation.

This idea has been popularised by the C++ template system, in which a *concept* embodies a set of operations necessary for a template to be instantiated. A concept is an implicit interface that specifies the operations necessary for instantiation. Consider this trivial C++ template function for equality:

```
template <typename A>
bool equals (A a1, A a2) {
    return a1 == a2;
}
```

The interface of the type parameter A is the implicit requirement that a value supports $operator==(.)$. In specific terms, this means that when the template for $equals\langle A \rangle()$ is expanded at the call site given a concrete type C (i.e. with the constraint that $A \equiv C$), the compiler tries to find a declaration for $operator==(c_1, c_2)$. If such a declaration is not found, the template cannot be instantiated. This is different from parametric polymorphism, because the template function performs an operation on values of type A , whereas a polymorphic function needs no knowledge of A values.

Unlike the implicit concept in C++ templates, Haskell supports the explicit specification of an interface using type class constraints [Wadler and Blott, 1989], which we illustrate with a function for determining equality on two lists. Equality is of course not restricted to a single type. In fact, many different datatypes support equality. But unlike the polymorphic $length$, equality must be implemented differently for each type a in $List\ a$, because the equality operation requires inspection of the elements in the list. The code below defines the class of types that support the equality ($==$) and inequality (\neq) operations.

```
class Eq a where
  (==), (≠) :: a → a → Bool
  a == b = not (a ≠ b)
  a ≠ b = not (a == b)
```

This type class definition includes the types of the interface operations and default implementations. For a datatype to support the operations in the class Eq , we create an instance of it. In our case, we use Boolean equality as we have defined it above.

```
instance Eq Bool where
  (==) = eqBool
```

Now we implement our list equality function in a generic manner and the instance $Eq\ Bool$ ensures that it works at least with Boolean values.

```
instance Eq a ⇒ Eq (List a) where
  Nil == Nil = True
  (Cons x xs) == (Cons y ys) = x == y ∧ xs == ys
  _ == _ = False
```

$List\ a$ is an instance of the Eq type class with the provided implementation of $==$. The $Eq\ a \Rightarrow$ part of the instance declaration is the *type-class context*. It imposes the constraint that the element type a supports the equality operation.

2.5 Property

Gibbons expands the concept of generic programming to include the properties or specifications of programs or program components. These are “generic”

in the sense that a specification may hold for multiple implementations. Properties may be informally or formally defined. Depending on the language or tool support, they may be encoded into a program, used as part of the testing process, or just appear as text.

A simple example of a property is found once again in the implementation of the equality function. A programmer with a classical logic view using an instance of the *Eq* type class above would likely expect that $x == y \equiv \text{not } (x \neq y)$ for any values of some type *a* such that $x, y :: Eq\ a \Rightarrow a$. However, there is no guarantee of this. Both $(==)$ and (\neq) are provided as separate methods, and the compiler cannot verify such a relationship. This informal law relies on programmers implementing the operations as expected for all instances of *Eq*.

Another example where generic properties are useful occurs with monads. The monad comes from category theory, and it was first applied to functional programming in order to structure “impure” features such as state, exceptions, and continuations [Wadler, 1990]. Since then, uses for the monad have expanded to include general input and output [Peyton Jones and Wadler, 1993], state threads [Launchbury and Peyton Jones, 1994], and parser combinators [Hutton and Meijer, 1998], among others.

In Haskell, a particular monad is defined as an instance of the following (simplified) *Monad* type class:

```
class Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

An instance must provide implementations for *return* and $(\gg=)$ (pronounced “bind”), the two primary operators used to structure monadic code. We use the *Maybe* datatype as an example:

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
  (Just x) ≫= k = k x
  Nothing ≫= _ = Nothing
  return      = Just
```

The *Maybe* datatype serves as a basic failure-tracking mechanism. Combined with the monadic structure, we can use it to string together multiple computations that might fail:

```
computeSomething :: Int → Maybe Int
computeSomething i = do j ← computeOrFail1 i
                    k ← do x ← computeOrFail2 j
                        y ← computeOrFail3 x
                    return y
                    return k
```

If any of the $computeOrFail_n :: Int \rightarrow Maybe\ Int$ functions fails with *Nothing*, the statements following that function are not executed and *computeSomething*

fails. Note that this example uses Haskell's **do**-notation which is a convenient syntactic sugar for repeated applications of ($\gg=$) [Peyton Jones, 2003].

The concept of the monad is not only specified by the type class *Monad*; it also requires that the functions satisfy a number of laws:

$$\begin{array}{lll} \text{return } a \gg= k & \equiv k \ a & \text{-- left unit} \\ m \gg= \text{return} & \equiv m & \text{-- right unit} \\ m \gg= (\lambda x \rightarrow k \ x \gg= h) & \equiv (m \gg= k) \gg= h & \text{-- associative} \end{array}$$

These three laws ensure that composition of monadic binds are associative and that *return* is both a left unit and a right unit of ($\gg=$) [Wadler, 1992]. All instances of the *Monad* class should obey these laws. While this property cannot be directly verified by the compiler, it is important for allowing support of the **do**-notation, especially nested **do** blocks as shown in *computeSomething* above.

2.6 Program Representation

There exist many techniques in which one program is parametrised by the representation of another program (or its own). This area includes such techniques as the following:

- Code generation, such as the generation of parsers and lexical analysers. Happy [Marlow and Gill, 1997] and Alex [Dornan et al., 2003] are Haskell programs for parser generation and lexical analysis, respectively.
- Reflection or the ability of a program to observe and modify its own structure and behavior. Reflection has been popularized by programming languages that support some dynamic type checking such as Java [Forman and Danforth, 1999], but some attempts have also been made in Haskell [Lämmel and Peyton Jones, 2004].
- Multi-stage programming for separating computation into stages [Taha, 1999].

Gibbons describes these ideas as genericity by stage; however, some techniques such as reflection do not immediately lend themselves to being staged. We think that this category of is better described as *metaprogramming* or generic programming in which the parameter is some representation of a program.

Perhaps the best known form of generic programming in this category is using C++ templates. The C++ template facility has led to a number of advanced libraries such as Boost [Gurtovoy and Abrahams, 2002] as well as other unexpected uses [Alexandrescu, 2001]. Partly inspired by C++ templates and the multi-stage programming language MetaML [Sheard, 1999], Template Haskell provides a metaprogramming extension to Haskell98 [Sheard and Peyton Jones, 2002].

We introduce the concept by example. An annoying itch in Haskell is the need to explicitly write selection functions for tuples of different arities. The standard library provides *fst* :: (a, b) → a and *snd* :: (a, b) → b, because pairs are

the most common form of tuples. But what about triples, quadruples, etc.? We can use Template Haskell to scratch that itch.

We want to automatically generate functions such as these:

```
fst3 = λ(x, -, -) → x
snd4 = λ(-, x, -, -) → x
```

Using Template Haskell, we can write:

```
fst3 = $(sel 1 3)
snd4 = $(sel 2 4)
```

This demonstrates the use of the “splice” syntax, $\$(\dots)$, to evaluate the enclosed “...” at compile time. Each call to $\$(sel\ i\ n)$ is expanded to a function that selects the i -th component of a n -tuple. Consider the following implementation³:

```
sel :: Int → Int → ExpQ
sel i n = lamE pats body
  where pats = [tupP (map varP as)]
        body = varE (as !! (i - 1))
        as   = [mkName ("a" ++ show j) | j ← [1..n]]
```

In order to define *sel*, we need to create an abstract syntax recipe of the form $\lambda(a_1, a_2, \dots, a_i, \dots, a_n) \rightarrow a_i$. We pull each of our ingredients from the Template Haskell libraries. First, we need a lambda expression (*lamE*) to create a function. A lambda expression requires a list of patterns on the left of the \rightarrow and an expression on the right. In our pattern, we use a tuple (*tupP*) and a list of n variables (*varP*) whose names are generated (using *mkName*) based on their position in the tuple. In the body of the lambda abstraction, we need an expression containing a variable (*varE*) that represents the i -th component. Finally, our function creates an expression of type *ExpQ* that can subsequently be evaluated within a splice.

Note that Template Haskell is type-safe and that a well-typed program cannot “go wrong” at run-time [Milner, 1978]. In the first stage of Template Haskell compilation, the expression inside the $\$(\dots)$ is type-checked. Next, this expression is compiled and executed, and the resulting piece of syntax is spliced into place where the call is made. Finally, the resulting Haskell program is completely type-checked as if this were again the initial stage of compilation. To put this into the context of our example above, compilation could fail for several reasons, namely the function *sel* does not type-check or a call to $\$(sel\ i\ n)$ generates code that does not type-check.

Template Haskell has also been explored for other uses in generic programming. Most notably, it is possible to prototype datatype-generic extensions to Haskell with Template Haskell [Norell and Jansson, 2004b].

³ The code for *sel* is derived from the original example [Sheard and Peyton Jones, 2002] with modifications to simplify it and to conform to the *Language.Haskell.TH* library included with GHC 6.8.2.

2.7 Shape

Recall the implementation of equality as defined by the type class *Eq*. Note that we gave only two instances, one for *Bool* and one for *List a*. In order for *Eq* to be useful, we need to provide a large number of instances: one for every primitive type and one for every algebraic datatype. While we cannot escape specialized definitions for basic types such as *Int* and *Char*, we should be able to abstract over the similarities of many algebraic datatypes. Intuitively, we can visualize these similarities by reviewing the syntax for datatypes in Haskell:

```
data List a = Nil | Cons a (List a)
```

From our understanding of the **data** declaration, we know that *List a* is constructed by either a *Nil* or a *Cons*. This choice may be called a *sum* and denoted by the $+$ symbol. We see that the *Cons* takes two arguments, *a* and another *List a*. We refer to this pair-like constructor as a *product* and use the \times symbol. A constructor that takes no arguments (such as *Nil*) also has the special designator of *unit*, often represented by **1** or $\mathbb{1}$. We can now reconstruct the *List* datatype in the new formulation:

```
data List a =  $\mathbb{1}$  + (a  $\times$  List a)
```

The *sum of products* view allows us to abstract over the shape or structure of a regular datatype. A *regular* datatype is a possibly recursive type whose definition does not involve a change of the type parameters. It can be formulated as an equation of sums, products, and units. For another, slightly more involved, example, take the following *Tree* datatype with its sum of products view:

```
data Tree a = Twig | Leaf a | Branch (Tree a) a (Tree a)
data Tree a =  $\mathbb{1}$  + (a + (Tree a  $\times$  (a  $\times$  Tree a)))
```

Tree has three alternatives: *Twig*, *Leaf*, and *Branch*. To simulate this, we use right-associative nested sums. Additionally, we use right-associative nested products to simulate the multiple argument *Branch* constructor. Using nesting simplifies the view, since the representation of any datatype only needs the nullary unit and the binary sum and product.

The sum of products view is used in a language extension such as *Generic Haskell* [Löh, 2004] to achieve datatype-generic programming. In fact, *Generic Haskell* provides a solution to the problem with our *Eq* type class mentioned above. Consider this implementation of datatype-generic equality:

```
eq{a :: *} :: (eq{a})  $\Rightarrow$  a  $\rightarrow$  a  $\rightarrow$  Bool
eq{Unit}   _      _      = True
eq{Int}    x      y      = eqInt x y
eq{Char}   x      y      = eqChar x y
eq{a + b}  (Inl x) (Inl y) = eq{a} x y
```

$$\begin{aligned}
eq\{a + b\} (Inl\ x) \ (Inr\ y) &= False \\
eq\{a + b\} (Inr\ x) \ (Inl\ y) &= False \\
eq\{a + b\} (Inr\ x) \ (Inr\ y) &= eq\{b\}\ x\ y \\
eq\{a \times b\} (x_1 \times y_1) (x_2 \times y_2) &= eq\{a\}\ x_1\ x_2 \wedge eq\{b\}\ y_1\ y_2
\end{aligned}$$

The $eq\{a :: \star\}$ function uses pattern matching on types to perform case analysis. Thus, each of the primitive types are specifically defined and the algebraic datatypes are supported with the generic sum of products view. We do not explain this function in detail, but rather rely on the reader's intuition for a cursory understanding. There are a number of resources for further study [Löh, 2004, Hinze and Jeuring, 2003a,b].

Generic Haskell (GH) is an example of a datatype-generic language extension to Haskell. However, GH requires a separate compiler/preprocessor to perform specialization for functions such as $eq\{a :: \star\}$ to the datatypes on which it is needed. For a comparison of GH and other approaches, see [Hinze et al., 2007]. In these lecture notes, we instead focus on library support for generic programming.

There are generic views other than the sum of products. For example, we may regard a datatype as a fixed point, allowing us to make all recursion in the datatype explicit. Another example is the spine view which we discuss in the Scrap Your Boilerplate library (Section 7). For a more in-depth study of generic views, refer to [Holdermans et al., 2006].

We have discussed a variety of generic programming techniques to provide background for the remainder of these lecture notes. Because a large part of programming involves the design of datatypes and the processing of values, one of the more interesting approaches is datatype-generic programming. As more and more programmers have used the Haskell language, they have developed a number of ways to structure data. Additionally, researchers have extended the language to increase its expressiveness. Before we jump into writing generic programs, we explore this world of Haskell datatypes in the next section.

3 The world of Haskell datatypes

Datatypes play a central rôle in programming in Haskell. Solving a problem often consists of designing a datatype, and defining functionality on that datatype. Haskell offers a powerful construct for defining datatypes: **data**. Haskell also offers two other constructs: **type** to introduce type synonyms and **newtype**, a restricted version of **data**.

Datatypes come in many variations: we have finite, regular, nested, and many more kinds of datatypes. This subsection introduces many of these variations of datatypes by example. Not all datatypes are pure Haskell 98, some require extensions to Haskell. Many of these extensions are supported by most Haskell compilers, some only by GHC. On the way, we explain kinds and show how they are used to classify types. For most of the datatypes we introduce, we define an equality function. As we will see, the definitions of equality on the

different datatypes follow a similar pattern. This pattern will also be used to define generic programs for equality in later sections covering LIGD (Section 5) and EMGM (Section 6).

3.1 Monomorphic datatypes

We start our journey through datatypes with lists containing values of a particular type. For example, in the previous section we have defined the datatype of lists of Booleans:

```
data ListB = NilB | ConsB Bool ListB
```

We define a new datatype, called List_B, which has two kinds of values: an empty list (represented by the constructor Nil_B), or a list consisting of a boolean value in front of another List_B. This datatype is the same as Haskell's predefined list datatype containing booleans, with [] and (:) as constructors. Since the datatype List_B does not take any type parameters, it has base kind *. Other examples of datatypes of kind * are Int, Char, etc.

Here is the equality function on this datatype:

```
eqListB :: ListB → ListB → Bool
eqListB NilB NilB = True
eqListB (ConsB b1 l1) (ConsB b2 l2) = eqBool b1 b2 ∧ eqListB l1 l2
eqListB - - = False
```

Two empty lists are equal, and two nonempty lists are equal if their head elements are the same (which we check using equality on Bool) and their tails are equal. An empty list and a nonempty list are unequal.

3.2 Polymorphic datatypes

We abstract from the datatype of Booleans in the type List_B to obtain polymorphic lists.

```
data List a = Nil | Cons a (List a)
```

Compared with List_B, the List a datatype has a different functional structure: the kind of List is * → *. Kinds represent the structure of datatypes, and are either * (base kind) or κ → ν, where κ and ν are kinds. Since List takes a base type as argument, it has the functional kind * → *. Actually, nothing is known about the type variable a, and in such a case, Haskell defaults its kind to *.

Equality on List is almost the same as equality on List_B.

```
eqList :: (a → a → Bool) → List a → List a → Bool
eqList eqa Nil Nil = True
eqList eqa (Cons x1 l1) (Cons x2 l2) = eqa x1 x2 ∧ eqList eqa l1 l2
eqList - - = False
```

The only difference with equality on List_B is that we need to have some means of determining equality on the elements of the list, so we need an additional equality function of type $(a \rightarrow a \rightarrow \text{Bool})$ as parameter⁴.

3.3 Families and mutually-recursive datatypes

A family of datatypes is a set of datatypes that may use each other. We can define a simplified representation of a system of linear equations using a non-recursive family of datatypes. A system of linear equations is a list of equations, each consisting of a pair linear expressions. For example, here is a system of three equations.

$$\begin{aligned}x - y &= 1 \\x + y + z &= 7 \\2x + 3y + z &= 5\end{aligned}$$

For simplicity, we assume linear expressions are values of a datatype for arithmetic expressions, $\text{Expr } a$. An arithmetic expression abstracts over the type of constants, typically an instance of the Num class, and is a variable, a literal, or the addition, subtraction, multiplication, or division of two arithmetic expressions.

```
type LinearSystem = List LinearExpr
data LinearExpr   = Equation (Expr Int) (Expr Int)
infixl 6 ×, ÷
infixl 5 +, -
data Expr a = Var String
            | Lit a
            | Expr a + Expr a
            | Expr a - Expr a
            | Expr a × Expr a
            | Expr a ÷ Expr a
```

Defining the equality function eq_{Expr} for LinearSystem is straightforward, and therefore we do not present it.

Datatypes in Haskell may also be mutually recursive, as can be seen in the following example. A forest is either empty or a tree followed by a forest, and a tree is either empty or a node of a forest:

```
data Tree a = Empty | Node a (Forest a)
data Forest a = Nil | Cons (Tree a) (Forest a)
```

Defining the equality function for these datatypes amounts to defining the equality function for each datatype separately. The result is a set of mutually recursive functions:

⁴ Using Haskell's type classes, this would correspond to replacing the type of the first argument in the type of eq_{List} by an $\text{Eq } a \Rightarrow$ constraint.

$$\begin{aligned}
eq_{Tree} &:: (a \rightarrow a \rightarrow Bool) \rightarrow Tree\ a \rightarrow Tree\ a \rightarrow Bool \\
eq_{Tree}\ eq_a\ Empty\ \quad &Empty \quad = True \\
eq_{Tree}\ eq_a\ (Node\ a_1\ f_1)\ (Node\ a_2\ f_2) &= eq_a\ a_1\ a_2 \wedge eq_{Forest}\ eq_a\ f_1\ f_2 \\
eq_{Tree}\ -\ \quad \quad \quad &\quad \quad = False
\end{aligned}$$

$$\begin{aligned}
eq_{Forest} &:: (a \rightarrow a \rightarrow Bool) \rightarrow Forest\ a \rightarrow Forest\ a \rightarrow Bool \\
eq_{Forest}\ eq_a\ Nil\ \quad \quad &Nil \quad = True \\
eq_{Forest}\ eq_a\ (Cons\ t_1\ f_1)\ (Cons\ t_2\ f_2) &= eq_{Tree}\ eq_a\ t_1\ t_2 \wedge eq_{Forest}\ eq_a\ f_1\ f_2 \\
eq_{Forest}\ -\ \quad \quad \quad &\quad \quad = False
\end{aligned}$$

Note that although the type `LinearSystem` defined previously uses several other types, it is not mutually recursive: `Expr a` is at the end of the hierarchy and is defined only in terms of itself.

3.4 Higher-order kinded datatypes

Consider the following minimal datatype for logic expressions, together with an equality function:

$$\begin{aligned}
\mathbf{data}\ Logic_s &= Lit\ Bool \\
&| Not\ Logic_s \\
&| Or\ Logic_s\ Logic_s \\
\\
eq_{Logic} &:: Logic_s \rightarrow Logic_s \rightarrow Bool \\
eq_{Logic}\ (Lit\ x_1)\ \quad (Lit\ x_2) &= eq_{Bool}\ x_1\ x_2 \\
eq_{Logic}\ (Not\ e_1)\ \quad (Not\ e'_1) &= eq_{Logic}\ e_1\ e'_1 \\
eq_{Logic}\ (Or\ e_1\ e_2)\ (Or\ e'_1\ e'_2) &= eq_{Logic}\ e_1\ e'_1 \wedge eq_{Logic}\ e_2\ e'_2 \\
eq_{Logic}\ -\ \quad \quad \quad &= False
\end{aligned}$$

Again, the equality function follows the pattern: for each pair of constructors with the same name, either recursively compare their arguments pairwise, or, if they do not take arguments, return `True`. Comparing different constructors results in `False`.

But suppose we now want to use the fact that disjunction is associative, and represent our `Logics` datatype as:

$$\begin{aligned}
\mathbf{data}\ Logic_L &= Lit\ Bool \\
&| Not\ Logic_L \\
&| Or\ (List\ Logic_L)
\end{aligned}$$

We can abstract from the “container type” `List`, which contains the subexpressions, by introducing a type argument for it.

$$\begin{aligned}
\mathbf{data}\ Logic_F\ f &= Lit\ Bool \\
&| Not\ (Logic_F\ f) \\
&| Or\ (f\ (Logic_F\ f))
\end{aligned}$$

We have introduced a type variable, and so Logic_F does not have kind \star as its predecessors. However, its kind is also not $\star \rightarrow \star$, as we have seen previously in the List datatype, because the type argument that Logic_F expects is not a base type, but a “type transformer”: we can see in the application f ($\text{Logic}_F f$) that f is applied to an argument. The kind of Logic_F is thus: $(\star \rightarrow \star) \rightarrow \star$. This datatype is a higher-order kinded datatype.

To better understand abstraction over container types, consider the following type:

type $\text{Logic}'_L = \text{Logic}_F \text{ List}$

Modulo undefined values, Logic'_L is isomorphic to Logic_L . The type argument of Logic_F describes what “container” will be used for the elements of the *Or* case.

Defining equality for the Logic'_L datatype is simple:

$$\begin{aligned} eq_{\text{Logic}'_L} &:: \text{Logic}'_L \rightarrow \text{Logic}'_L \rightarrow \text{Bool} \\ eq_{\text{Logic}'_L} (\text{Lit } x_1) (\text{Lit } x_2) &= eq_{\text{Bool}} x_1 x_2 \\ eq_{\text{Logic}'_L} (\text{Not } x_1) (\text{Not } x_2) &= eq_{\text{Logic}'_L} x_1 x_2 \\ eq_{\text{Logic}'_L} (\text{Or } l_1) (\text{Or } l_2) &= \\ &\quad \text{length } l_1 == \text{length } l_2 \wedge \text{and } (\text{zipWith } eq_{\text{Logic}'_L} l_1 l_2) \\ eq_{\text{Logic}'_L} - - &= \text{False} \end{aligned}$$

Note that we use the $\text{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow \text{List } a \rightarrow \text{List } b \rightarrow \text{List } c$ function, because we know the container is the list type.

We can also provide the equality function for the Logic_F type:

$$\begin{aligned} eq_{\text{Logic}_F} &:: (\forall a. (a \rightarrow a \rightarrow \text{Bool}) \rightarrow f a \rightarrow f a \rightarrow \text{Bool}) \rightarrow \\ &\quad \text{Logic}_F f \rightarrow \text{Logic}_F f \rightarrow \text{Bool} \\ eq_{\text{Logic}_F} eq_f (\text{Lit } x_1) (\text{Lit } x_2) &= eq_{\text{Bool}} x_1 x_2 \\ eq_{\text{Logic}_F} eq_f (\text{Not } x_1) (\text{Not } x_2) &= eq_{\text{Logic}_F} eq_f x_1 x_2 \\ eq_{\text{Logic}_F} eq_f (\text{Or } x_1) (\text{Or } x_2) &= eq_f (eq_{\text{Logic}_F} eq_f) x_1 x_2 \\ eq_{\text{Logic}_F} - - &= \text{False} \end{aligned}$$

This definition is considerably more complex than the previous one. The added complexity is caused by the fact that we do not know what the “container” type is, and therefore we have to abstract from an equality function on such a type. However, since that type is just a container, its equality function also needs to abstract from an equality function on the contained type! This requires the use of rank-2 polymorphism, a common extension to Haskell, and is indicated by the presence of the $\forall a$ in the type signature of eq_{Logic_F} . When invoking this function, the user supplies the first argument (the equality function on the *f*-type).

3.5 Nested datatypes

A *regular* data type (as introduced in Section 2.7) is a possibly recursive, parametrised type whose recursive occurrences do not involve a change of type parameters. All the datatypes we have introduced so far are regular. However, it is also possible to define so-called nested datatypes [Bird and Meertens, 1998], in which recursive occurrences of the datatype may have other type arguments than the datatype being defined. Perfectly balanced binary trees are an example of such a datatype.

```
data Perfect a = Leaf a | Node (Perfect (a, a))
```

Any value of this datatype is a full binary tree in which all leaves are at the same depth. This is attained by using the pair constructor in the recursive call for the *Node* constructor. An example of such tree is:

```
perfect = Node (Node (Node (Leaf (((1,2), (3,4))), ((5,6), (7,8))))))
```

Here is the equality function on *Perfect*:

```
eqPerfect :: (∀a . a → a → Bool) → Perfect b → Perfect b → Bool
eqPerfect eqa (Leaf x1) (Leaf x2) = eqa x1 x2
eqPerfect eqa (Node x1) (Node x2) = eqPerfect eqa x1 x2
eqPerfect _ _ _ = False
```

This definition is again very similar to the equality on datatypes we have introduced before. An interesting aspect of this definition is that it needs rank-2 polymorphism (as in *eqLogicP*), since the type of the elements at the leaves depends on how deep the tree is.

3.6 Existentially quantified datatypes

Many of the datatypes we have seen take arguments, and in the type of the constructors of these datatypes, those type arguments are universally quantified. For example, the constructor *Cons* of the datatype *List a* has type $a \rightarrow List\ a \rightarrow List\ a$ for all types *a*. However, we can also use existential types, which “hide” a type variable that only occurs under a constructor. Consider the following example:

```
data Dynamic = ∀a . Dyn (Rep a) a
```

The type *Dynamic* encapsulates a type *a* and its representation, a value of type *Rep a*. We will encounter the datatype *Rep a* later in these lecture notes (Section 5), where it is used to convert between datatypes and their run-time representations. Despite the use of the \forall symbol, the type variable *a* is said to be existentially quantified because it is only available inside the constructor—*Dynamic* has kind \star . Existential datatypes are typically used to encapsulate some type

with its corresponding actions: in the above example, the only thing we can do with a `Dynamic` is to inspect its representation. Other important applications of existentially quantified datatypes include the implementation of abstract datatypes, which encapsulate a type together with a set of operations. Existential datatypes are not part of the Haskell 98 standard, but they are a fairly common extension.

The definition of equality on an existentially quantified datatype may be problematic. We can only compare two values if the operations provided by the constructor allow us to compare them. For example, if the only operation provided by the constructor is a string representation of the value, we can only compare the string representation of two values. Therefore equality can only be defined as the equality of the visible components of the existential datatype.

3.7 Generalized algebraic datatypes

Another powerful extension to the Haskell 98 standard are generalized algebraic datatypes (GADTs). A GADT is a datatype in which different constructors may have related but different result types. Consider the following example, where we combine the datatypes `Logics` and `Expr` shown before in a datatype for statements:

```
data Stat a where
  Val  :: Expr Int    → Stat (Expr Int)
  Term :: Logics     → Stat Logics
  If   :: Stat Logics → Stat a → Stat a → Stat a
  Write :: Stat a     → Stat ()
  Seq  :: Stat a      → Stat b → Stat b
```

The new aspect here is the ability to give each constructor a different result type `Stat...`. This has the advantage that we can describe the type of the different constructors more precisely. For example, the type of the `If` constructor now says that the first argument of the `If` returns a logic statement, and the statements returned in the “then” and “else” branches may be of any type, as long as they have the same type.

Defining equality of two statements is still a matter of repeating similar code:

```
eqStat :: Stat a → Stat b → Bool
eqStat (Val x1) (Val x2) = eqExpr (==) x1 x2
eqStat (Term x1) (Term x2) = eqLogic x1 x2
eqStat (If x1 x2 x3) (If x'1 x'2 x'3) = eqStat x1 x'1 ∧ eqStat x2 x'2 ∧ eqStat x3 x'3
eqStat (Write x1) (Write x2) = eqStat x1 x2
eqStat (Seq x1 x2) (Seq x'1 x'2) = eqStat x1 x'1 ∧ eqStat x2 x'2
eqStat _ _ = False
```

We have shown many varieties of datatypes and the example of the equality function, which offers functionality needed on many datatypes. We have

seen that we can define the equality functions ourselves, but the code quickly becomes repetitive and tedious. Furthermore, if a datatype changes, the definition of the equality function has to change accordingly. This is not only inefficient and time-consuming but also error-prone. The generic programming libraries introduced in the rest of these lecture notes will solve this problem.

4 Libraries for generic programming

Datatype-generic programming has been around for more than 10 years now. The first approaches to datatype-generic programming used programming language extensions to describe generic functions. Since the beginning of this decade quite a number of libraries for generic programming in Haskell have been developed. The rationale for developing a library for generic programming instead of a language extension is that Haskell is powerful enough to support most generic programming concepts by means of a library. Furthermore, compared with a language extension, a library is much easier to ship, support, and maintain. A library might be accompanied by tools that depend on non-standard language extensions, for example for generating embedding-projection pairs, but the core is Haskell.

The libraries for generic programming have different characteristics. Recently, an extensive comparison of generic programming libraries (and their characteristics) has been performed [Rodriguez et al., 2008b]. In these notes we will discuss three of those libraries: a Lightweight Implementation of Generics and Dynamics, Extensible and Modular Generics for the Masses, and Scrap Your Boilerplate.

We focus on these three libraries for a number of reasons. First, we think these libraries are representative examples: one library explicitly passes a type representation as argument to a generic function, another relies on the type class mechanism, and the third is traversal- and combinator-based. Furthermore, all three have been used for a number of generic functions, and are relatively easy to use for parts of the lab exercise given in these notes. Finally, all three of them can express many generic functions; the Uniplate library [Mitchell and Runciman, 2007] is also representative and easy to use, but Scrap Your Boilerplate is more powerful.

The example libraries show different ways to implement the essential ingredients of generic programming libraries. Support for generic programming consists of three essential ingredients: a run-time type representation, a generic view on data, and support for overloading.

A *type-indexed function* (TIF) is a function that is defined on every type of a family of types. We say that the types in this family index the TIF, and we call the type family a universe. The run-time representation of types determines the universe on which we can pattern match in a type-indexed function. The larger this universe, the more types the function can be applied to.

A type-indexed function only works on the universe on which it is defined. If a new datatype is defined, the type-indexed function cannot be used on this

new datatype. There are two ways to make it work on the new datatype. A non-generic extension of the universe of a TIF requires a type-specific, ad-hoc case for the new datatype. A generic extension (or a generic view) of a universe of a TIF requires to express the new datatype in terms of the universe of the TIF so that the TIF can be used on the new datatype without a type-specific case. A TIF combined with a generic extension is called a *generic function*.

An overloaded function is a function that analyses types to exhibit type-specific behavior. Type-indexed and generic functions are special cases of overloaded functions. Many generic functions even have type-specific behavior: lists are printed in a non-generic way by the generic pretty-printer defined by deriving *Show* in Haskell.

In the next sections we will see how to encode these basic ingredients in the three libraries we introduce. For each library, we present its run-time type representation, the generic view on data and how overloading is achieved.

5 Lightweight Implementation of Generics and Dynamics

This section introduces generic programming in a simplified form of the datatype-generic programming library Lightweight Implementation of Generics and Dynamics (LIGD, [Cheney and Hinze, 2002]).

5.1 An example function

In polymorphic lambda calculus it is impossible to write a single parametrically polymorphic equality function that works on all datatypes [Wadler, 1989]. That is why the definition of equality in Haskell uses type classes, and ML uses equality types. The *Eq* type class provides the equality operator `=`, which is overloaded for a family of types. To add a newly defined datatype to this family, the programmer defines an instance of equality for it. Thus, the programmer manually writes definitions of equality for every new datatype that is defined, as we did in Section 2. For equality, this process could be automated by using the type class deriving mechanism. However, this mechanism can only be used with a small number of type classes, because it is hardwired into the language, making it closed and impossible to extend or change by the programmer. In this subsection we show how equality is defined once and for all datatypes in LIGD.

The equality function *eq* takes three arguments: the two values of type `a` to compare, and a representation of the type of these values `Rep a`.

$$eq :: \text{Rep } a \rightarrow a \rightarrow a \rightarrow \text{Bool}$$

Function *eq* is defined by pattern matching on the representation type `Rep`. The representation type contains a constructor for the unit type, which represents types with a single value, for the sum type, which represents a choice between two types, and for the product type, which represents a pair of types. We will

introduce the type `Rep` and its constructors in the following subsection. Function `eq` is an example of a TIF.

```

eq (RUnit)    Unit    Unit    = True
eq (RInt )    i      j      = i == j
eq (RChar)    c      d      = c == d
eq (RSum r_a r_b) (L a_1) (L a_2) = eq r_a a_1 a_2
eq (RSum r_a r_b) (R b_1) (R b_2) = eq r_b b_1 b_2
eq (RSum r_a r_b) -      -      = False
eq (RProd r_a r_b) (a_1 :×: b_1) (a_2 :×: b_2) = eq r_a a_1 a_2 ∧ eq r_b b_1 b_2

```

Note that the run-time type-representation argument of function `eq` of type `Rep a` represents the type of the following two arguments, namely `a`. The type `Unit`, represented by `RUnit`, contains a single value `Unit`. Since there is only a single value of type `Unit`, two values of the type are always equal. For the `Int` type and the `Char` type we use the equality functions that are available for these types. For the `:+:` type we first check if the two values have the same outermost constructor (`L` or `R`). If so, we recursively compare their arguments, using the type representation of the type of those arguments. For the product type `:×:` we pairwise compare the components of the product, using the representation types of the types of these components. Function `eq` is type-safe, in the sense that the two arguments that we want to compare have the same type, given by the run-time type representation `Rep a`. If an argument would have another type, Haskell's type-checker would complain at compile-time.

5.2 Run-time type representation

In LIGD, the first argument of a type-indexed function is a run-time *type representation*, which describes the type of the the arguments, results, or both arguments and results, of the function. The type representation need not appear as the first argument, but this is standard practice. In the variant of LIGD we give in these notes, types are represented by a generalized algebraic data type. Using a GADT has the advantage that case analysis on types can be implemented by pattern matching, a familiar construct to functional programmers. The GADT represents the types of the universe consisting of units, sums and products, together with basic types such as integers and characters. Of course, there are many more basic types, such as floats, doubles, etc, but we only include `Int` and `Char`.

```

infixr 5 :+ :
infixr 6 :× :

```

```

data Unit = Unit
data a :+ : b = L a | R b
data a :× : b = a :×: b

```

Unit is an example of a type with a single constructor with no arguments. The `:+` datatype is equal to the datatype `Either` in Haskell's prelude. The `:×` equals the pair (tuple) type in Haskell. We have chosen to use new datatypes to represent type representations at run-time, to not mix the world of type representations and other types. The GADT `Rep` uses these datatypes to represent the structure of types.

```

data Rep t where
  RUnit ::                Rep Unit
  RInt  ::                Rep Int
  RChar ::                Rep Char
  RSum  :: Rep a → Rep b → Rep (a :+: b)
  RProd :: Rep a → Rep b → Rep (a :×: b)

```

The original LIGD was developed before GADTs had been added to GHC and used an existentially quantified datatype instead. GADTs make it easier to define these structure types.

5.3 Going generic: universe extension

If we define a datatype, how can we use our type-indexed function on this new datatype? In LIGD (and most other generic programming libraries), the introduction of a new datatype does not require redefinition or extension of all existing generic functions. We merely need to describe the new datatype to the library, and all existing and future generic functions will be able to handle it.

In LIGD, the structure of a datatype `b` is represented by the following `Rep` constructor.

```
RType :: EP b c → Rep c → Rep b
```

The type `c` is the *structure representation* type of `b`, where `c` is a type isomorphic to `b`. The isomorphism is witnessed by an embedding projection pair, which is a pair of functions that convert `b` values to `c` values and back.

```
data EP b c = EP {from :: (b → c), to :: (c → b)}
```

In LIGD, constructors are represented by nested sum types and constructor arguments are represented by nested product types. The structure representation type for lists is `Unit :+: a :×` `List a`, and the embedding projection for lists is as follows:

```

fromList :: List a → Unit :+: a :× List a
fromList Nil           = L Unit
fromList (Cons a as)  = R (a :× as)
toList :: Unit :+: a :× List a → List a
toList (L Unit)      = Nil
toList (R (a :× as)) = Cons a as

```

Note that the components of the pair are not embedded in the universe. The reason for this is that LIGD does not model recursion explicitly.

To extend the universe to lists, we write a type representation using *RType*:

$$r_{List} :: \text{Rep } a \rightarrow \text{Rep } (\text{List } a)$$

$$r_{List} r_a = RType \left(EP \text{ from}_{List} \text{ to}_{List} \right) \\ \left(RSum \text{ RUnit } (RProd r_a (r_{List} r_a)) \right)$$

We defined a type-indexed function for equality above. This definition still misses a case to handle datatypes that are represented by *RType*. The definition of this case is given below. It takes two values, transforms them to their structure representations and applies equality to the resulting values.

$$eq (RType ep r_a) t_1 t_2 = eq r_a (\text{from } ep t_1) (\text{from } ep t_2)$$

Adding this line to the definition of *eq* turns it into a generic function.

In summary, there are two ways to extend a universe to a type *T*. A non-generic extension requires type-specific, ad-hoc cases for *T* in type-indexed functions, and a generic-extension requires a structure representation of *T* but no additional function cases. This is the feature that distinguishes type-indexed functions and generic functions. The latter include a case for *RType*, which allows them to exploit the structure of a datatype in order to apply generic uniform behaviour to values of that datatype; while the former do not have a case for *RType*, and therefore rely exclusively on non-generic extension.

In LIGD, sums, products and units are used to represent the structure of a datatype. Certainly other choices are possible. For example, PolyLib includes the datatype *Fix* in the universe, in order to represent the recursive structure of datatypes. We refer to these representation choices as *generic views* [Holdermans et al., 2006]. Informally, a view consists of base (or view) types for the universe (for example *:+:* and *:×:*) and a convention to represent structure, for example, representing constructors by nested sums. Usually the choice of a view will have an impact on the expressiveness of a library, that is, which generic functions definitions are supported and what are the set of datatypes on which generic extension is possible.

Exercise 1. Give the representation of the datatypes *Tree* and *Forest* in terms of *Rep*. ■

5.4 Support for overloading

Function *eq* can be viewed as an implementation of **deriving** *Eq* in Haskell itself. Similarly, we can define functions that implement the methods of the other classes that can be derived in Haskell: *Show*, *Read*, *Ord*, *Enum*, and *Bounded*. We will show a definition of a generic show function. Besides illustrating generic programming in the library in general, implementing a generic show function illustrates how a library deals with constructor names, and how a library deals with ad-hoc cases for particular datatypes. For example, we do

not want a generic show function to show a string "abc" as a list with explicit occurrences of constructor names: $Cons\ 'a'\ (Cons\ 'b'\ (Cons\ 'c'\ Nil))$.

The type representations as discussed in the previous section do not contain any information about constructors, and hence it is impossible to define a generic show function using this representation. To deal with constructor names, we add an extra constructor to the structure representation type.

$$RCon :: String \rightarrow Rep\ a \rightarrow Rep\ a$$

Using this extra constructor of Rep, the representation of the datatype List a becomes:

$$\begin{aligned} r_{List} &:: Rep\ a \rightarrow Rep\ (List\ a) \\ r_{List}\ r_a &= RType\ (EP\ from_{List}\ to_{List}) \\ &\quad (RSum\ (RCon\ "Nil"\ RUnit) \\ &\quad\quad (RCon\ "Cons"\ (RProd\ r_a\ (r_{List}\ r_a)))) \end{aligned}$$

Here is a first attempt at defining a generic show function:

$$\begin{aligned} show &:: Rep\ t \rightarrow t \rightarrow String \\ show\ RInt\ \quad t &= show\ t \\ show\ RChar\ \quad t &= show\ t \\ show\ RUnit\ \quad t &= "" \\ show\ (RSum\ r_a\ r_b)\ (L\ a) &= show\ r_a\ a \\ show\ (RSum\ r_a\ r_b)\ (R\ b) &= show\ r_b\ b \\ show\ (RProd\ r_a\ r_b)\ (a\ :\times\ b) &= show\ r_a\ a\ ++\ " "\ ++\ show\ r_b\ b \\ show\ (RType\ ep\ r_a)\ t &= show\ r_a\ (from\ ep\ t) \\ show\ (RCon\ s\ RUnit)\ t &= s \\ show\ (RCon\ s\ r_a)\ \quad t &= "("\ ++\ s\ ++\ " "\ ++\ show\ r_a\ t\ ++\ ")" \end{aligned}$$

This definition works for all datatypes, but it shows strings and Haskell's lists in a uniform way, using constructor names. We want to extend this function so that it behaves in a special, non-generic way for the type of strings and the List a datatype.

For each type for which we want a generic function to behave in a special, non-generic, way, we have to extend the representation type. For example, to solve the problem with showing strings and lists, we add the following constructors to the representation type Rep:

$$\begin{aligned} RString &:: Rep\ String \\ RList &:: Rep\ a \rightarrow Rep\ (List\ a) \end{aligned}$$

Now we can add the following lines to the generic show function to obtain type-specific behavior for the type String and List a.

$$\begin{aligned} show\ (RList\ r_a)\ Nil &= "[]" \\ show\ (RList\ r_a)\ (Cons\ x\ xs) &= show\ r_a\ x\ ++\ ":\ " ++\ show\ (RList\ r_a)\ xs \\ show\ RString\ s &= s \end{aligned}$$

The resulting function does not implement all details of **deriving** *Show*, but it does provide the core functionality.

Note that we had to adapt the type representation type *Rep* to obtain type-specific behavior in the *gshow* function. It is undesirable to adapt a library for the purpose of obtaining special behavior of a single generic function on a particular datatype. Unfortunately, this is unavoidable in the LIGD library: for any generic function that needs special behavior on a particular datatype, we have to extend the type representation with that datatype. This implies that many users will construct their own variant of the LIGD library, making both the library and the generic functions written using it less portable and reusable. Löh and Hinze [2006] show how to add *open datatypes* to Haskell. A datatype is open if it can be extended in a different module. In a language with open datatypes, the above problem with LIGD would disappear.

5.5 Generic functions in LIGD

This section introduces some more generic functions in LIGD, in particular some functions for which we need different type representations. We start with a simple example of a generic program.

Empty

With every type we can associate an empty value. For example, the empty value of type *Int* is 0, and the empty value of type *List a* is *Nil*. Function *empty* is a generic function that returns the empty value for an arbitrary type. An interesting aspect of this function is that it *constructs* a value of a type, instead of consuming a value, as in *eq* and *show*.

```
empty :: Rep a → a
empty RUnit      = Unit
empty RInt       = 0
empty RChar      = '\NUL'
empty (RSum ra rb) = L (empty ra)
empty (RProd ra rb) = empty ra :×: empty rb
empty (RType ep ra) = to ep (empty ra)
empty (RCon s ra) = empty ra
empty _          = undefined
```

Exercise 2. Another generic function that constructs values of a datatype is the function *enum* :: *Rep a* → [*a*], which generates all values of a type. Many datatypes have infinitely many values, so it is important that function *enum* enumerates values fairly. Implement *enum* in LIGD. ■

Flatten

Many datatypes can be considered “container” datatypes: datatypes used to store and structure values. Examples are the datatypes `List a`, `Tree a`, `Forest a`, `Expr a`, all introduced in Section 3. One of the few datatypes introduced in Section 3 that is not a container datatype is the datatype `Logics`. A common function on a container datatype is the function *flatten*, which takes a value of the datatype, and returns a list containing all values that it contains. For example, on the datatype `Tree a` function *flatten* would have type `Tree a → [a]`. This subsection explains how the generic *flatten* function is defined in LIGD.

To implement function *flatten*, we have to solve a number of problems. A first problem is to describe its type. A first, incorrect, attempt would be the following:

$$\textit{flatten} :: \text{Rep } f \rightarrow f \ a \rightarrow [a]$$

where `f` abstracts over types of kind $\star \rightarrow \star$. Since `Rep` has kind $\star \rightarrow \star$, this gives a kind error. Replacing `Rep f` by `Rep (f a)` would solve the kinding problem, but introduces another: how do we split the representation of a container datatype into a representation for `f` and a representation for `a`? Type application is implicit in our type representation. We solve this problem by adapting the structure representation type for LIGD with an extra case *RVar1* which is used to define special functionality for occurrences of the type argument in constructors.

```
data Rep1 f a where
  RUnit1 ::                               Rep1 f Unit
  RSum1  :: Rep1 f a → Rep1 f b → Rep1 f (a :+: b)
  ...
  RVar1  :: f a →                               Rep1 f a
```

Except for its type, the representation of lists using this new representation type does not change.

```
rList,1 :: Rep1 f a → Rep1 f (List a)
rList,1 ra = RType1 (EP fromList toList)
               (RSum1 (RCon1 "Nil" RUnit1)
                    (RCon1 "Cons" (RProd1 ra (rList,1 ra))))
```

To obtain an instance of the generic function *flatten* on the datatype `List a` we write:

```
flattenList :: List a → [a]
flattenList = flattenT rList,1
```

To specify the type of *flattenT* we introduce a **newtype** `Flatten`, together with an abbreviation for a representation type `Rep1` in which the action for type variables has been instantiated with `Flatten`:

```

newtype Flatten b a = Flatten { gFlatten :: a → b }
type    a ⇒ b      = Rep1 (Flatten b) a

```

which is used in specifying what to do with an occurrence of a value of the type variable. Function *mkFlatten* specifies what to do with such an occurrence: store it in a list by means of the function $(:[])$.

```

mkFlatten :: a ⇒ [a]
mkFlatten = RVar1 (Flatten (:[]))

```

Function *flatten* now takes a function which transforms a representation of the action on variables $a ⇒ [a]$ to a representation of the argument datatype $b ⇒ [a]$, and a value of the argument datatype b , and returns the list of occurrences of the values of the type variable $[a]$.

```

flattenT :: ((a ⇒ [a]) → (b ⇒ [a])) → b → [a]
flattenT repTransform = flatten (repTransform mkFlatten)

```

where *flatten* is the generic function that does the pattern matching on the structure representation type.

```

flatten :: b ⇒ [a] → b → [a]
flatten RUnit1      Unit      = []
flatten (RSum1 ra rb) (L a)  = flatten ra a
flatten (RSum1 ra rb) (R b)  = flatten rb b
flatten (RProd1 ra rb) (a :×: b) = flatten ra a ++ flatten rb b
flatten RInt1       i         = []
flatten RChar1      c         = []
flatten (RCon1 _ ra) x       = flatten ra x
flatten (RType1 ep ra) x     = flatten ra (from ep x)
flatten (RVar1 f)   x         = gFlatten f x

```

Note that using $++$ in the product case makes function *flatten* rather inefficient. The number of reduction steps needed is quadratic in the length of the returned list. Using a standard accumulation technique this can be turned into a linear function. We omit this definition.

Exercise 3. Many generic functions follow the pattern of the generic *flatten* function. Examples include a function that sums all the integers in a value of a datatype, and a function that takes the Boolean “or” of all Boolean values in a value of a datatype. We implement this pattern with *crushr*.

The function *crushr* abstracts over functionality at occurrences of the type variable. In the definition of *flatten*, this includes the base case $[]$ and the binary case $++$. The relevant types of *crushr* follow.

```

newtype Crush b a = Crush { gCrush :: a → b }
crushr :: Rep1 (Crush b) a → (b → b → b) → b → a → b

```

Define *crushr*. (Attempt to solve it without looking ahead to Section 6 in which *crushr* is defined using the EMGM library.)

To test if your function implements the desired behavior, instantiate *crushr* with the addition operator, 0, and a value of a datatype containing integers to obtain a generic *sum* function. ■

Generalised map

A well-known generic function is the generic *map* function, which is a generalisation of the *map* function available on lists in Haskell. The *map* function in Haskell takes a function and a list as argument, and applies the function to all elements in the list. The generic *map* function takes a function of type $a \rightarrow b$ and a value of a datatype containing *a*'s, and applies the function to all the *a*'s in the value. Function *map* can be viewed as the implementation of **deriving** for the *Functor* type class in Haskell. Just as function *flatten*, the generic *map* function needs to know where the occurrences of the type-argument of the datatype appear in a constructor. This implies that we have to be able to abstract over type constructors. If we use the representation type *Rep1* to implement the generic *map* function, we can only define a *map* function in which the argument function returns a value of either a type that only depends on *a*, or a constant type. This is because the constructor *RVar1* has type $f\ a \rightarrow \text{Rep1}\ f\ a$. We use the *RVar1* constructor to specify the behavior at occurrences of values of the type variable, and to specify a function of type $a \rightarrow b$ there, we need an extra type variable. Since we want to be able to change the type of the values occurring in a datatype using the generic *map* function, we change the representation datatype once more.

data *Rep2* *f* *a* *b* **where**

```

RUnit2 ::                               Rep2 f Unit Unit
RSum2  :: Rep2 f a b → Rep2 f c d → Rep2 f (a :+: c) (b :+: d)
...
RVar2  :: f a b →                               Rep2 f a b

```

The only difference with the representation type *Rep1* is the occurrence of the extra type variable *b* in the representation type and all constructors. The representation of lists using this new representation type hardly changes: we get an extra embedding-projection pair for the extra type variable.

$$r_{List,2} r_a = RType2 \left(\begin{array}{l} EP\ from_{List}\ to_{List} \\ EP\ from_{List}\ to_{List} \\ RSum2\ (RCon2\ "Nil"\ RUnit2) \\ RCon2\ "Cons"\ (RProd2\ r_a\ (r_{List,2}\ r_a)) \end{array} \right)$$

Using $r_{List,2}$, function *map* on lists is obtained as follows:

```

mapList :: (a → b) → List a → List b
mapList = map r_{List,2}

```

where *map* is defined below. To define function *map* we follow the same approach as for defining *flatten*. First we introduce the type *Map* for functions, and the type $a \rightarrow b$ for a mapping in the representation type

newtype $\text{Map } a \ b = \text{Map}\{g\text{Map} :: a \rightarrow b\}$
type $(a \rightarrow b) = \text{Rep2 } \text{Map } a \ b$

Function *mkMap* specifies what to do with an occurrence of a value of the type variable, namely applying the argument function.

$mkMap :: (a \rightarrow b) \rightarrow (a \rightarrow b)$
 $mkMap \ f = \text{RVar2 } (\text{Map } f)$

Function *map* now takes a function which transforms a representation of the action on variables $a \rightarrow b$, a function, and a value of a datatype, and returns a value of the same shape, in which the argument function has been applied to all values appearing at the type variable position in the constructors.

$map :: ((a \rightarrow b) \rightarrow (c \rightarrow d)) \rightarrow (a \rightarrow b) \rightarrow (c \rightarrow d)$
 $map \ \text{repTransform } f = \text{mapG_LIGD } (\text{repTransform } (mkMap \ f))$

where *mapG_LIGD* is the generic function that does the pattern matching on the structure representation type.

Note that if we pass the representation transformer $r_{List,2}$ to *map*, the type of its first argument is unified to $a \rightarrow b$:

$r_{List,2} :: (a \rightarrow b) \rightarrow (\text{List } a \rightarrow \text{List } b)$

Applying *map* to $r_{List,2}$ results in a function of the desired type $(a \rightarrow b) \rightarrow \text{List } a \rightarrow \text{List } b$.

$mapG_LIGD :: (a \rightarrow b) \rightarrow a \rightarrow b$

$mapG_LIGD \ \text{RUnit2}$	Unit	$= \text{Unit}$
$mapG_LIGD \ (\text{RSum2 } r_a \ r_b)$	$(L \ a)$	$= L \ (mapG_LIGD \ r_a \ a)$
$mapG_LIGD \ (\text{RSum2 } r_a \ r_b)$	$(R \ b)$	$= R \ (mapG_LIGD \ r_b \ b)$
$mapG_LIGD \ (\text{RProd2 } r_a \ r_b)$	$(a \ :\times\ b)$	$= mapG_LIGD \ r_a \ a \ :\times\ mapG_LIGD \ r_b \ b$
$mapG_LIGD \ \text{RInt2}$	i	$= i$
$mapG_LIGD \ \text{RChar2}$	c	$= c$
$mapG_LIGD \ (\text{RCon2 } _ \ r_a)$	x	$= mapG_LIGD \ r_a \ x$
$mapG_LIGD \ (\text{RType2 } ep_1 \ ep_2 \ r_a)$	x	$= (\text{to } ep_2 \ . \ mapG_LIGD \ r_a \ . \ \text{from } ep_1) \ x$
$mapG_LIGD \ (\text{RVar2 } f)$	x	$= g\text{Map } f \ x$

Since the last two generic functions we have introduced both require a new structure representation type, one might wonder if this would happen for many generic functions. As far as we are aware, these extensions stop at level 3. We could use the datatype *Rep3* for all of our generic functions, but that would introduce many type variables which are never used, so we prefer to use the representation type that is most suitable for the generic function at hand.

Exercise 4. Define the generalised version of function $zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$ in LIGD. You may adapt the structure representation type for this purpose. ■

5.6 Case study: exercise assistants

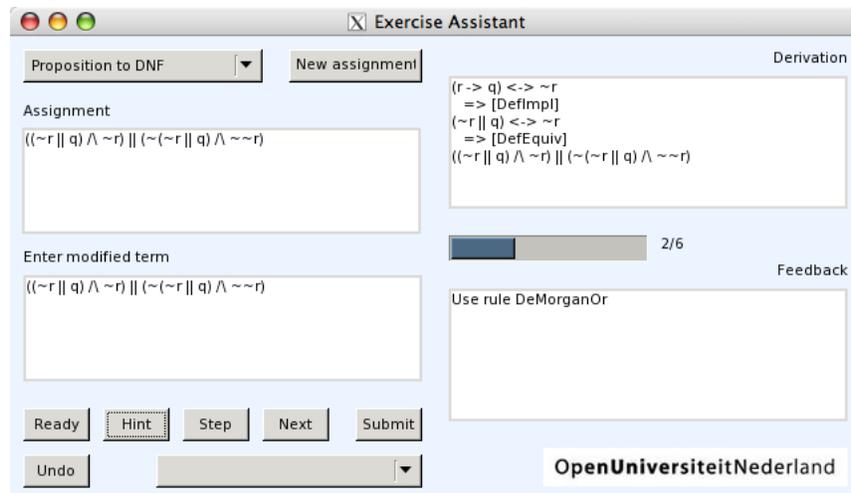


Fig. 1. The Exercise Assistant

An exercise assistant supports interactively solving exercises in a particular domain. For example, at the Open University NL and Utrecht University, we are developing an exercise assistant that supports interactively solving a system of linear equations [Passier and Jeuring, 2006], an assistant that supports calculating a disjunctive normal form (DNF) of a logical expression [Lodder et al., 2006], and an assistant that supports several kinds of exercises within linear algebra. A screenshot of the assistant that supports calculating a DNF of a logical expression is shown in Figure 1.

The exercise assistants for the different domains are very similar: they need functionality for rewriting, equality, exercise generation, term traversals, selections, serialization, etc. The different exercise assistants can be viewed as an instance of a generic program. For each generic programming library we discuss in these lecture notes, we will discuss how to implement a part of the exercise assistant in the library. In this subsection we will show how to implement a generic function for determining the difference between two terms.

The methods from the *Eq* class, $=$ and \neq , determine whether or not two values are equal. Often we do not just want to know whether or not two values are equal, but if they differ, *how much* or *where* they differ. For example, in the

exercise assistant a user can submit a step towards a solution of an exercise. We want to compare the submitted expression against expressions obtained by applying rewrite rules to the previous expression. If none of these match, we want to find a correctly rewritten expression that is closest in some sense to the expression submitted by the student. We now define a function *measureEqual* that determines a measure of equality between two values. Given two values, the function counts the number of constructors and basic values that are equal. The function traverses its arguments top-down; as soon as it encounters unequal constructors, it does not further traverse into these values.

```

measureEqual :: Rep a -> a -> a -> Int
measureEqual RUnit      -      -      = 1
measureEqual RInt       i      j      = if i == j then 1 else 0
measureEqual RChar      c      d      = if c == d then 1 else 0
measureEqual (RSum ra rb) (L l) (L r) = measureEqual ra l r
measureEqual (RSum ra rb) (R l) (R r) = measureEqual rb l r
measureEqual (RSum ra rb) l      r      = 0
measureEqual (RProd ra rb) (l :×: r) (s :×: t) =
  measureEqual ra l s + measureEqual rb r t
measureEqual (RType ep ra) l      r      =
  measureEqual ra (from ep l) (from ep r)
measureEqual (RCon s ra) l      r      = 1 + measureEqual ra l r

```

If we also have a definition of a function *gsize* which returns the size of a value, counting all basic values for one, we define the function *gdiff* by:

```

gdiff :: Rep a -> a -> a -> Int
gdiff repr r l = size repr r - measureEqual repr r l

```

Function *measureEqual* gives a rough equality measure. A generic minimum edit distance program would give a more precise equality measure.

6 Extensible and Modular Generics for the Masses

This section introduces generic programming as described by “Generics for the Masses” [Hinze, 2006], in particular the extensible and modular variant (“EMGM”) introduced by Oliveira et al. [2006].

Before beginning the description of EMGM, we should note that, while writing these lecture notes, we created the first release of EMGM as a packaged library [Utrecht, 2008] for general consumption. We realized that, having done a large amount of work for explanatory purposes, it would be a pity if there was not a real body of code that people could use.

As it turned out, the process of writing, documenting, and testing a library also contributed something back to the writing of these notes. We established a uniform notation (for both LIGD and EMGM) and even adapted the design of EMGM to fix a sore spot (see Section 6.6). We can recommend combining

the complementary tasks of writing an article and releasing a library to anyone for whom the situation applies. Each task has a different set of problems to be solved, but we think that together both result in better research and better code.

In the following sections, we try as much as possible to use the same notation and approaches used in the distribution; however, there are numerous occasions where we simplify example code for didactic purposes. For example, to see a more efficient and extensive generic *show*, refer to the library code.

Information about the EMGM library can be found at <http://www.cs.uu.nl/wiki/GenericProgramming/EMGM>.

6.1 An example function

Defining a generic function in the EMGM library involves several steps. First, we declare the type signature of a function in a **newtype** declaration.

```
newtype Eq a = Eq { selEq :: a → a → Bool }
```

selEq is a label for a function value wrapped in the type *Eq*. It is important to remember that record labels also serve as destructors and implicitly expect a record value as the first argument. The actual type of *selEq* is:

```
selEq :: Eq a → a → a → Bool
```

A value of *Eq a* contains an instance of the equality function for a representation of a type *a*. We use *selEq* to select the instance in a way similar to *eq* in LIGD (Section 5.1):

```
selEq (rprod rchar rint) ('Q' :×: 42) ('Q' :×: 42) ~ True
```

By passing the representation to the type-indexed function *selEq*, we specialize it to the datatype representation. The following functions serve as cases for each supported type:

```
selEqint   i      j      = i == j
selEqchar  c      d      = c == d
selEq1     Unit   Unit   = True
selEq+ ra rb (L a1) (L a2) = selEq ra a1 a2
selEq+ ra rb (R b1) (R b2) = selEq rb b1 b2
selEq+ - - - - -         = False
selEq× ra rb (a1 :×: b1) (a2 :×: b2) = selEq ra a1 a2 ∧ selEq rb b1 b2
```

We can read this in the same fashion as a type-indexed function in LIGD. Indeed, there is a high degree of similarity. Instead of a single function that uses pattern matching on type representation, however, we have many functions, each corresponding to a primitive or a structural type. Another major difference with LIGD is that the the type representation parameters are explicit and

not embedded in the `Rep` datatype. Specifically, each function takes the appropriate number of representations according to its arity. For example, `selEq1` has no representation arguments, and `selEq+` and `selEq×` each have two.

These functions are only part of the story, however. Notice that `selEq+` and `selEq×` each call the function `selEq`. We need to tie the recursive knot, so that `selEq` will select the appropriate case. We do this by creating an instance declaration for the **newtype** `Eq`:

```
instance Generic Eq where
  rint      = Eq selEqint
  rchar     = Eq selEqchar
  runit     = Eq selEq1
  rsum ra rb = Eq (selEq+ ra rb)
  rprod ra rb = Eq (selEq× ra rb)
```

The type class `Generic` has member functions corresponding to primitive and structure types. Each method defines the instance of the type-indexed function for the associated type. Our type case functions are now used to construct values of `Eq`. The EMGM approach uses method overriding instead of the pattern matching used by LIGD, but it still provides an effective case analysis on types.

We now have all of the necessary parts to use the type-indexed function `selEq`; however, we should not need to provide an explicit representation every time. So, we introduce a convenient wrapper that determines which type representation we need.

```
eq :: (Rep a) => a -> a -> Bool
eq = selEq rep
```

We discuss the `Rep` class in more detail in the next section. For now, we may think of the class context here simply as a requirement that type `a` be representable. The value `rep` is a dispatcher to that representation.

6.2 Run-time type representation

In contrast with LIGD's GADT, EMGM makes extensive use of type classes for its run-time type representation. The primary classes are `Generic` and `Rep`, though others may be used to extend the basic concepts of EMGM as we will see later (Section 6.3).

The class `Generic` serves as the type case for generic functions.

```
infixr 5 'rsum'
infixr 6 'rprod'
class Generic g where
  rint  :: g Int
  rchar :: g Char
  runit :: g Unit
```

```

rsum :: g a → g b → g (a :+: b)
rprod :: g a → g b → g (a :×: b)

```

The class is parametrised by the type constructor `g` with the intention that `g` is a **newtype** representing the type of a generic function. We saw this used in the `selEq` example.

Each method of the class represents a case of the generic function. The function supports the same universe of types as LIGD (e.g. `Unit`, `:+:`, `:×:`, and primitive types). Also like LIGD, the structural induction is implemented through recursive calls, but unlike LIGD, these are polymorphically recursive. Thus, in our previous example, each call to `selEq` may have a different type.

The generic function as we have defined it to this point is a deconstructor for the type `g`. As such, it requires an instance of `g`, the type representation. In order to alleviate this requirement, we use another type class:

```

class Rep a where
  rep :: (Generic g) ⇒ g a

```

The class `Rep` allows us to replace any instance of the type `g a` with `rep`. This simple but powerful concept uses the type system to dispatch the necessary representation. This representation is again built inductively using the methods of `Generic`:

```

instance Rep Int where
  rep = rint
instance Rep Char where
  rep = rchar
instance Rep Unit where
  rep = runit
instance (Rep a, Rep b) ⇒ Rep (a :+: b) where
  rep = rsum rep rep
instance (Rep a, Rep b) ⇒ Rep (a :×: b) where
  rep = rprod rep rep

```

As simple as these instances of `Rep` are, they handle an important duty. In the function `eq`, we use `rep` to instantiate the structure of the arguments. For example, it instantiates `rprod rchar rint` with the argument `'Q' :×: (42 :: Int)`. Now, we may apply `eq` with the same ease of use as with any ad-hoc polymorphic function, even though it is actually datatype-generic.

6.3 Going generic: universe extension

Much like in LIGD, we need to extend our universe to include any new datatypes that we create. We extend our type-indexed functions with a case to support arbitrary datatypes.

class *Generic* g **where**

...
 $rtype :: EP\ b\ a \rightarrow g\ a \rightarrow g\ b$

The *rtype* function reuses the embedding-projection pair datatype EP mentioned earlier to witness the isomorphism between the structure representation and the datatype. Note the similarity with the *RType* constructor from LIGD.

To demonstrate the use of *rtype*, we will once again show how the List datatype may be represented in a value. As mentioned before, we use the same structure types as LIGD, so we can make use of the same pair of functions, $from_{List}$ and to_{List} , in the embedding projection for lists. Using this pair and an encoding of the list structure at the value level, we define a representation of lists:

$$r_{List} :: (Generic\ g) \Rightarrow g\ a \rightarrow g\ (List\ a)$$
$$r_{List}\ r_a = rtype\ (EP\ from_{List}\ to_{List})\ (runit\ 'rsum'\ r_a\ 'rprod'\ r_{List}\ r_a)$$

It is now straightforward to apply a generic function to a list. To make it convenient, we create a new instance of *Rep* for List a with the constraint that the contained type a must also be representable:

instance (*Rep* a) \Rightarrow *Rep* (List a) **where**
 $rep = r_{List}\ rep$

At last, we can transform our type-indexed equality function into a true generic function. For this, we need to add another case for arbitrary datatypes.

$$selEq_{type}\ ep\ r_a\ a_1\ a_2 = selEq\ r_a\ (from\ ep\ a_1)\ (from\ ep\ a_2)$$

instance *Generic* Eq **where**

...
 $rtype\ ep\ r_a = Eq\ (selEq_{type}\ ep\ r_a)$

The function $selEq_{type}$ accepts any datatype for which an embedding-projection pair has been defined. It is very similar to the *RType* case in the LIGD version of equality. The *Generic* instance definition for *rtype* completes the requirements necessary for *eq* to be a generic function.

Exercise 5. Now that you have seen how to define r_{List} , you should be able to define the representation for most other datatypes. Give representations and embedding-projection pairs for $Logic_L$ and $Logic_F$ from Section 3. You may need to do the same for other datatypes in the process. Test your results using *eq* as defined above. ■

6.4 Support for overloading

In this section, we demonstrate how the EMGM library supports constructor names and ad-hoc cases. As with LIGD, we illustrate this support using a generic *show* function and lists and strings.

Currently, we cannot access information such as constructor names in the definition of a generic function. For this purpose, we add another case to our generic function declaration.

```
class Generic g where
  . . .
  rcon :: String → g a → g a
```

We use *rcon* to label other structure components with a constructor name⁵. As an example of using this method, we modify the list type representation with constructor names:

```
rList :: (Generic g) ⇒ g a → g (List a)
rList ra = rtype (EP fromList toList)
              (rcon "Nil" runit 'rsum' rcon "Cons" (ra 'rprod' rList ra))
```

Using the capability to display constructor names, we can write a simplified generic show function⁶:

```
newtype Show a = Show { selShow :: a → String }

selShowint      i      = show i
selShowchar    c      = show c
selShow1       Unit    = ""
selShow+      ra rb (L a) = selShow ra a
selShow+      ra rb (R b) = selShow rb b
selShow×      ra rb (a :×: b) = selShow ra a ++ " " ++ selShow rb b
selShowtype ep ra  a      = selShow ra (from ep a)
selShowcon s  ra  a      = "(" ++ s ++ " " ++ selShow ra a ++ ")"

instance Generic Show where
  rint      = Show selShowint
  rchar     = Show selShowchar
  runit     = Show selShow1
  rsum  ra rb = Show (selShow+ ra rb)
  rprod ra rb = Show (selShow× ra rb)
  rtype ep ra = Show (selShowtype ep ra)
  rcon  s  ra = Show (selShowcon s ra)
```

⁵ Note that the released EMGM library uses `ConDescr` instead of `String`. `ConDescr` contains a more comprehensive description of a constructor (e.g. fixity, arity, etc.). For simplicity's sake, however, we only use a `String` for the constructor name here.

⁶ The example of the generic *show* is greatly simplified for didactic purposes. Refer to the released library for a more involved and efficient implementation of *show*.

```
show :: (Rep a) => a -> String
show = selShow rep
```

Applying this function to a list of integers gives us the expected result:

```
show (Cons 5 (Cons 3 Nil)) ~> "(Cons 5 (Cons 3 (Nil )))"
```

As mentioned in Section 5, we would prefer to see this list appear as it natively does in Haskell: "[5,3]". We would also rather see the String type appear as a row of characters instead of a list with constructor names. To this end, just as we added constructors to the Rep GADT, we can add methods to the *Generic* type class.

```
class Generic g where
  ...
  rlist  :: g a -> g (List a)
  rstring :: g String
```

It is then straightforward to define these new cases for the generic show function.

```
selShowlist ra Nil          = "[]"
selShowlist ra (Cons a as) = selShow ra a ++ ":" ++ selShow (rlist ra) as
selShowstring                = id
```

```
instance Generic Show where
  ...
  rlist ra = Show (selShowlist ra)
  rstring = Show selShowstring
```

Our last step is to make these types representable. We replace the previous instance of *Rep* for List *a* with one using the *rlist* method, and we add a new instance for String.

```
instance (Rep a) => Rep (List a) where
  rep = rlist rep
instance Rep String where
  rep = rstring
```

Now, when we revisit the example application of *show* above, we receive a more concise response: "5:3: []". We leave final refinement of the function (i.e. printing "[5,3]") as an exercise for the reader.

In order to extend the generic function representation to support ad-hoc list and string cases, we modified the *Generic* type class. This approach fails when the module containing *Generic* is distributed as a third-party library. Users of the library must choose to either modify the library itself or duplicate the functionality in their own programs. Fortunately, there are solutions to making a Generics for the Masses library extensible and modular.

6.5 Making generic functions extensible

Since modifying the type class *Generic* should be considered off-limits, we might consider declaring a hierarchy of classes for extensibility. *Generic* would then be the base class for all generic functions. A user of the library would introduce a subclass for an ad-hoc case on a datatype. To explore this idea, let us revisit the example of defining a special case for *show* on lists.

The subclass for list appears as follows:

```
class (Generic g) ⇒ GenericList g where  
  rlist :: g a → g (List a)  
  rlist = rList
```

This declaration introduces the class *GenericList* encoding a list representation. The default value of *rlist* is the same value that we determined previously, but it can be overridden in an instance declaration. For the ad-hoc case of the generic *show* function, we would use an instance with the same implementation as before:

```
instance GenericList Show where  
  rlist r = Show (selShowlist r)
```

We have regained some ground on our previous implementation of an ad-hoc case, yet we have lost some as well. We can apply our generic function to a type representation and a value (e.g. (*selShow* (*list rint*) (*Cons* 3 *Nil*))), and it will evaluate as expected. However, we can no longer use the same means of dispatching the appropriate representation with ad-hoc cases. What happens if we attempt to write the following instance of *Rep*?

```
instance (Rep a) ⇒ Rep (List a) where  
  rep = rlist rep
```

GHC returns with this error:

```
Could not deduce (GenericList g)  
  from the context (Rep (List a), Rep a, Generic g)  
  arising from a use of ‘rlist’ at ...  
Possible fix:  
  add (GenericList g) to the context of  
  the type signature for ‘rep’ ...
```

We certainly do not want to follow GHC’s advise. Recall that the method *rep* of class *Rep* has the type (*Generic* g, *Rep* a) ⇒ g a. By adding *GenericList* g to its context, we would force all generic functions to support both *Generic* and *GenericList*, thereby ruling out any modularity. In order to use *Rep* as it is currently defined, we must use a type g that is an instance of *Generic*; instances of any subclasses are not valid.

Let us instead abstract over the type constructor *g*. We subsequently re-define *Rep* as a type class with two parameters.

```
class Rep g a where  
  rep :: g a
```

This migrates the parametrisation of the type constructor to the class level and lifts the restriction of the *Generic* context. We now re-define the representative instances.

```
instance (Generic g) ⇒ Rep g Int where  
  rep = rint  
instance (Generic g) ⇒ Rep g Char where  
  rep = rchar  
instance (Generic g) ⇒ Rep g Unit where  
  rep = runit  
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (a :+: b) where  
  rep = rsum rep rep  
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (a :×: b) where  
  rep = rprod rep rep  
instance (GenericList g, Rep g a) ⇒ Rep g (List a) where  
  rep = rlist rep
```

The organization here is very regular. Every instance handled by a method of *Generic* is constrained by *Generic* in its context. For the ad-hoc list instance, we use *GenericList* instead.

Now, we rewrite our generic show function to use the new dispatcher by specialising the type constructor argument *g* to *Show*.

```
show :: (Rep Show a) ⇒ a → String  
show = selShow rep
```

6.6 Reducing the burden of extensibility

The approach of using a type-specific class for extensibility [Oliveira et al., 2006] puts an extra burden on a user of the library. Without the change for extensibility (i.e. before Section 6.4), a function such as *show* in EMGM would automatically work with any type that was an instance of *Rep*. When we add Section 6.5, then every generic function must have an instance of every datatype that it will support. In other words, even if we did not want to define an ad-hoc case for *Show* using *GenericList* as we did earlier, we must provide at least the following instance if we use *show* on lists.

```
instance GenericList Show where
```

This uses the default method for *rlist* and overrides nothing.

As developers of a library, we want to strike a balance between ease of use and flexibility. Since we want to allow for extensibility in EMGM, we cannot provide these instances for each generic function provided by the library. This forces the library user to write one for every unique pair of datatype and generic function that is used, whether or not an ad-hoc case is desired. We can fortunately reduce this burden using an extension to Haskell semantics.

Overlapping instances allow more than one instance declaration to match when resolving the class context of a function, provided that there is a most specific one. Using overlapping instances, we no longer need a type-specific class such as *GenericList*, because constraint resolution will choose the list representation as long as *List a* is the most specific instance.

To continue with our example of specializing *Show* for lists, we provide the changes needed with respect to Section 6.5. The *List* instance for *Rep* is the same except for replacing *GenericList* with *Generic*.

```
instance (Generic g, Rep g a)  $\Rightarrow$  Rep g (List a) where  
  rep = rlist rep
```

At this point, with overlapping instances enabled, no further work is necessary for lists to be supported by any generic function that uses the *Generic* class. However, since we do want an ad-hoc case, we add an instance for *Show*:

```
instance (Rep Show a)  $\Rightarrow$  Rep Show (List a) where  
  rep = Show (selShowlist rep)
```

Notice that the **newtype** *Show* is substituted for the variable *g* in the first argument of *Rep*.

It is straightforward to also define an ad-hoc case for *String*.

```
instance Rep Show String where  
  rep = Show selShowstring
```

In the previous extensibility approach, we would have needed an additional *GenericString* class.

Exercise 6. The standard *compare* function returns the ordered relationship (“less than,” “equal to,” or “greater than”) between two instances of some type *a*.

```
data Ordering = LT | EQ | GT  
compare :: (Eq a, Ord a)  $\Rightarrow$  a  $\rightarrow$  a  $\rightarrow$  Ordering
```

This function can be implemented by hand, but more often it is generated by the compiler using **deriving** *Ord*. The latter uses the syntactic ordering of constructors to determine the relationship. For example, the datatype *Ordering* derives *Ord* and its constructors have the relationship *LT* < *EQ* < *GT*.

Implement an extensible, generic version of *compare* that behaves like **deriving** *Ord* and works with representable types. It should have a type signature similar to the above, but with a different class context. ■

6.7 Generic functions in EMGM

In this section, we discuss the implementation of various generic functions. Some require alternative strategies from the approach described so far.

Empty

Up to now, we have only seen generic “consumer” functions, or functions that accept generic arguments. A “producer” function returns a generic value. A simple example of a generic producer is one that generates an “empty” value for every possible type. We write the generic empty function in EMGM as follows:

```
newtype Empty a = Empty { selEmpty :: a }
```

```
instance Generic Empty where
```

```
  rint      = Empty 0
  rchar     = Empty '\NUL'
  runit     = Empty Unit
  rsum      ra rb = Empty (L (selEmpty ra))
  rprod     ra rb = Empty (selEmpty ra :×: selEmpty rb)
  rtype ep ra = Empty (to ep (selEmpty ra))
  rcon s ra = Empty (selEmpty ra)
```

```
empty :: (Rep Empty a) ⇒ a
empty = selEmpty rep
```

There are two noteworthy differences from previous examples. First, notice the deconstructor, $selEmpty :: Empty \rightarrow a$, which takes a representation and produces a value; no other arguments are provided. In order to generate an empty value, we need to specify a concrete type:

```
empty :: Int :+: Char ~> L 0
```

The other difference lies in the *rtype* definition, where we use *to ep* instead of *from ep*. This is characteristic of a producer function.

Crush and flatten

We have seen generic functions abstracted over finite and saturated types (with kind \star), but EMGM also supports type constructors (kind $\star \rightarrow \star$). There are many such “container” datatypes using parametric polymorphism.

We explore the realm of type constructors with the implementation of a generic crush function. Crush is a fold-like operation over a datatype [Meertens, 1996]. It is a very flexible function, and many other useful functions can be implemented using crush. We demonstrate this with a generic flatten operation.

We want to end up with a function similar to this:

```
crushr :: (a -> b -> b) -> b -> f a -> b
```

The function *crushr* takes three arguments: a “combining” operator that joins a-values with b-values to create new b-values, a “zero” value, and a container *f* of a-values. *crushr* (sometimes called *reduce*) is a generalization of the standard Haskell right-fold, *foldr*. In *foldr*, *f* is specialized to `[]`.

We split the implementation of *crushr* into components, and we begin with the type signature for the combining function.

```
newtype Crushr b a = Crushr { selCrushr :: a -> b -> b }
```

This function extracts the container’s element and combines it with a partial result to produce a final result. The implementation follows⁷:

```
crushrint      -      e = e
crushrchar    -      e = e
crushr1       -      e = e
crushr+      ra rb (L a)  e = selCrushr ra a e
crushr+      ra rb (R b)  e = selCrushr rb b e
crushr×      ra rb (a :×: b) e = selCrushr ra a (selCrushr rb b e)
crushrtype ep ra a      e = selCrushr ra (from ep a) e
crushrcon s ra a      e = selCrushr ra a e
```

```
instance Generic (Crushr b) where
```

```
  rint = Crushr crushrint
```

```
  . . .
```

Note that *selCrushr* is only applied to the parametrised structural type cases: *crushr₊*, *crushr_×*, *crushr_{type}*, and *crushr_{con}*; it is not applied to the primitive types. Crush only combines the elements of a polymorphic datatype and does not act on non-parametrised types.

We have successfully made it this far, but now we run into a problem. The type for *rep* is *Rep g a => g a*, and type *a* is the representation type and has kind \star . We need a representation for a type of kind $\star \rightarrow \star$. To expand *rep* to support type constructors, we define similar class in which the method has a parameter.

```
class FRep g f where
```

```
  frep :: g a -> g (f a)
```

The class *FRep* (representation for functionally kinded types) takes the same first type argument as *Rep*, but the second is the type constructor *f*. Notice that

⁷ For brevity, we elide most of the instance declaration, because it is as expected.

the type of *frep* matches the kind of *f*. This is exactly what we need for types such as *Tree* or *List*. The *FRep* instance for *List* is not too unlike the one for *Rep*:

```
instance (Generic g) => FRep g List where
  frep = rList
```

To actually define *crushr* is now possible; however, it is a bit of a puzzle to put the pieces together. Let's review what we have to work with.

```
Crushr :: (a -> b -> b) -> Crushr b a
frep :: (FRep g f) => g a -> g (f a)
selCrushr :: Crushr b a -> a -> b -> b
```

Applying some analysis of the types (left as an exercise for the reader), we compose these functions to get our result.

```
selCrushr . frep . Crushr :: (FRep (Crushr b) f) => (a -> b -> b) -> f a -> b -> b
```

Finally, we rearrange the arguments to get a final product with a signature similar to *foldr*.

```
crushr :: (FRep (Crushr b) f) => (a -> b -> b) -> b -> f a -> b
crushr f z x = selCrushr (frep (Crushr f)) x z
```

To demonstrate the use of *crushr*, we use the same generic flattening function introduced in Section 5. Recall that flattening involves translating all elements of a structure into a list. The definition of *flattenr* requires only the combining operator, *(:)*, and the zero value, *[]*.

```
flattenr :: (FRep (Crushr [a]) f) => f a -> [a]
flattenr = crushr (:) []
```

Exercise 7. Why does the function name of *crushr* end with an *r*? What difference would you expect from a function called *crushl*? ■

Exercise 8. Define two functions using *crushr*:

1. *showElements* takes a container with showable elements and returns a string with the elements printed in a comma-delimited fashion.
2. *sumElements* takes a container with numeric elements and returns the numeric sum of all elements.

■

Generalised map

The standard library *map* function applies a function to every element in a list. A datatype-specific map function may be implemented as an instance of the standard class *Functor*; however, this approach is error-prone. A generic map operation generalises *map* to any container datatype.

As described in Section 5.5, a generic *map* function gives us the ability to modify the elements of any container type. We aim for a function with this type:

$$\text{map} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

As with *crushr*, we first introduce the function that applies to each element.

```
newtype Map a b = Map { selMap :: a → b }
```

selMap appears to be similar to *selCrushr*, so we might expect to implement *map* like *crushr*. The difference lies in the fact that *crushr* is only a generic consumer of a datatype while *map* is both a consumer and a producer (as described in Section 6.7). We need to abstract over both type arguments in *Map*.

The implementation of the generic function follows:

$$\begin{aligned} \text{map}_{\text{int}} \quad x &= x \\ \text{map}_{\text{char}} \quad x &= x \\ \text{map}_{\mathbb{1}} \quad x &= x \\ \text{map}_{+} \quad r_a\ r_b\ (L\ a) &= L\ (\text{selMap}\ r_a\ a) \\ \text{map}_{+} \quad r_a\ r_b\ (R\ b) &= R\ (\text{selMap}\ r_b\ b) \\ \text{map}_{\times} \quad r_a\ r_b\ (a\ :\times\ b) &= \text{selMap}\ r_a\ a\ :\times\ \text{selMap}\ r_b\ b \\ \text{map}_{\text{type}}\ ep_1\ ep_2\ r_a\ a &= (\text{to}\ ep_2.\ \text{selMap}\ r_a.\ \text{from}\ ep_1)\ a \end{aligned}$$

Since this function is also a generic producer, the definitions create new values with the same representation. Additionally, *map_{type}* uses both *from* and *to* for converting between datatypes and the structural representation. We need only one representation, *r_a*, to map over, but we need two different embedding-projection pairs, *ep₁* and *ep₂*, to translate between the input type, the representation, and the output type.

In order to support abstraction over two types, we need a new class for defining generic functions. One option is to add a type argument to *Generic* and reuse that type class for all previous implementations, ignoring the extra variable. Instead, for simplicity, we choose to create *Generic2* to distinguish generic functions with two type arguments.

```
class Generic2 g where
  rint2  :: g Int Int
  rchar2 :: g Char Char
  runit2 :: g Unit Unit
  rsum2  :: g a1 a2 → g b1 b2 → g (a1 :+: b1) (a2 :+: b2)
```

$$\begin{aligned} rprod2 &:: g\ a_1\ a_2 \rightarrow g\ b_1\ b_2 \rightarrow g\ (a_1\ :\times\ b_1)\ (a_2\ :\times\ b_2) \\ rtype2 &:: EP\ a_2\ a_1 \rightarrow EP\ b_2\ b_1 \rightarrow g\ a_1\ b_1 \rightarrow g\ a_2\ b_2 \end{aligned}$$

Again, it is a simple matter to make `Map` an instance of `Generic2`:

```
instance Generic2 Map where
  rint2           = Map map_int
  ...
  rtype2 ep1 ep2 r_a = Map (map_type ep1 ep2 r_a)
```

Now that `rtype2` differs from `rtype`, we need to recreate the representation of datatypes. Fortunately, the difference is the minor addition of another embedding-projection pair. We write the representation for list as:

$$\begin{aligned} r_{List,2} &:: (Generic2\ g) \Rightarrow g\ a\ b \rightarrow g\ (List\ a)\ (List\ b) \\ r_{List,2}\ r_a &= rtype2\ (EP\ from_{List}\ to_{List})\ (EP\ from_{List}\ to_{List}) \\ &\quad (runit2\ 'rsum2'\ r_a\ 'rprod2'\ r_{List,2}\ r_a) \end{aligned}$$

We can immediately use the list representation to implement the standard `map` as `mapList`:

```
mapList :: (a -> b) -> List a -> List b
mapList = selMap . r_{List,2} . Map
```

Of course, our goal is to generalise this, but we need an appropriate dispatcher class. `FRep` will not work, because it abstracts over only one type variable. We need to extend it just as we extended `Generic` to `Generic2`:

```
class FRep2 g f where
  frep2 :: g a b -> g (f a) (f b)
instance (Generic2 g) => FRep2 g List where
  frep2 = r_{List,2}
```

The class `FRep2` uses a type constructor `g` representing a generic function with two argument types. Note, however, that we still expect functionally kinded datatypes: `f` has kind $\star \rightarrow \star$.

Finally, we provide our definition of `map`.

```
map :: (FRep2 Map f) => (a -> b) -> f a -> f b
map = selMap . frep2 . Map
```

This definition follows as the expected generalisation of `mapList`.

Exercise 9. Are there other useful generic functions that make use of `Generic2` and/or `FRep2`? Can you define them? ■

Exercise 10. Define a generalisation of the standard function `zipWith` in EMGM. The result should have a type signature similar to this:

$$zipWith :: (a \rightarrow b \rightarrow c) \rightarrow f\ a \rightarrow f\ b \rightarrow f\ c$$

What extensions to the library (as defined) are needed? ■

6.8 Case study: generation

The exercise assistant offers the possibility to generate a new exercise for a student. This implies that we need a set of exercises for every domain: systems of linear equations, logical expressions, etc. We can create this set either by hand for every domain or generically for an arbitrary domain. The former would likely involve a lot of work, much of which is duplicated for each domain. For the latter, we need to randomly generate exercises. This leads us to defining a generic random value generator.

At the simplest, we seek a function with this type signature:

```
gen :: Int → a
```

gen takes an integer and returns a value somehow representative of that number. The integer can be randomly generated. Suppose that for small `Int` arguments (e.g. greater than 0 but single-digit), *gen* produces relatively simple values (e.g. with few sums). Then, as the number increases, the output is more and more complex. This would lead to an output like QuickCheck [Claessen and Hughes, 2000] typically uses for testing. It would also lead to a set of exercises that progressively get more difficult as they are solved.

One approach to doing this is to enumerate the values of a datatype. We generate a list of all of the values using the following function template:

```
newtype Enum a = Enum { selEnum :: [a] }
instance Generic Enum where
  rint      = Enum enum_int
  rchar     = Enum enum_char
  runit     = Enum enum_1
  rsum  r_a r_b = Enum (enum_+   r_a r_b)
  rprod  r_a r_b = Enum (enum_×   r_a r_b)
  rtype ep r_a  = Enum (enum_type ep r_a)
  rcon  s r_a   = Enum (enum_con s r_a)
```

Now, let's look at each case. `Int` values can be positive or negative and cover the range from *minBound* to *maxBound*, the exact values of these being dependent on the implementation. A simple option might be:

```
enum_int = [0 .. maxBound :: Int] ++ [minBound .. (-1)]
```

However, that would lead to a (very long) list of positive numbers followed by another (very long) list of negative numbers. This is an awfully unbalanced sequence. For the sake of enumeration, we would prefer to start with the most "basic" value (equivalent to the *empty* value in Section 6.7) and alternate positive and negative numbers.

```
enum_int = [0 .. maxBound :: Int] ||| ((-1) 'downTo' minBound)
where downTo n m | n < m = []
      downTo n m      = n : downTo (n - 1) m
```

By reversing the negative enumeration, we now begin with 0 and go larger. We define the `|||` operator as follows:

```
(|||) :: [a] -> [a] -> [a]
[]      ||| ys = ys
(x : xs) ||| ys = x : ys ||| xs
```

The operator is similar to `++` with the exception of the recursive case, in which `xs` and `ys` are swapped. This allows us to interleave the elements of the two lists, thus balancing the positive and negative sides of the `Int` enumeration. Note that this also works fine for infinite lists.

For `Char` and `Unit`, the implementations are straightforward.

```
enum_char = [minBound .. maxBound :: Char]
enum_1    = [Unit]
```

We designate the most basic character as the first, so we only need to enumerate the characters from `minBound` to `maxBound`. `Unit` has only one value, so it has a single-element list.

In sums, we have a problem analogous to that of `Int`. We want to generate `L`-values and `R`-values, but we want to choose fairly from each side.

```
enum_+ r_a r_b = [L x | x <- selEnum r_a] ||| [R y | y <- selEnum r_b]
```

By interleaving these lists, we ensure that there is no preference to either alternative.

We use the Cartesian product for enumerating the pairs of two (possibly infinite) lists.

```
enum_x r_a r_b = selEnum r_a <X> selEnum r_b
```

The definition of `<X>` is left as an exercise for the reader.

The cases for `Enum` are `enum_type` and `enum_con`. The former requires a `map` to convert a list of generic representations to a list of values. The latter is the same as for `Empty` (Section 6.7), because constructor information is not used here.

```
enum_type ep r_a = map (to ep) (selEnum r_a)
enum_con s r_a = selEnum r_a
```

The final step for a generic enumeration function is to apply it to a representation.

```
enum :: (Rep Enum a) => [a]
enum = selEnum rep
```

To get to a generic generator, we simply index into the list.

```
gen :: (Rep Enum a) => Int -> a
gen = (!) enum
```

Exercise 11. Define a function that takes the diagonalization of a list of lists.

$$diag :: [[a]] \rightarrow [a]$$

diag returns a list of all of elements in the inner lists. It will always return at least some elements from every inner list, even if that list is infinite.

We can then use *diag* to define the Cartesian product.

$$\begin{aligned} (\times) &:: [a] \rightarrow [b] \rightarrow [a \times b] \\ xs \times ys &= diag \ [[x \times y \mid y \leftarrow ys] \mid x \leftarrow xs] \end{aligned}$$

■

Exercise 12. Design a more efficient generic generator function. ■

7 Scrap Your Boilerplate

In this section we describe the Scrap Your Boilerplate (SYB) approach to generic programming [Lämmel and Peyton Jones, 2003, 2004]. The novel concept behind this approach to generic programming is that contrary to the two approaches discussed previously in these notes, the structure of datatypes is not directly exposed to the programmer. In the SYB approach, generic functions are defined in terms of “primitive” generic combinators, which do not have to be given by the user because they are derivable using Haskell’s **deriving** mechanism for type classes.

7.1 An example function

Recall the `Expr` datatype from Section 3), and suppose we want to implement a function that increases the value of each literal by one. Here is a simple but incorrect solution:

$$\begin{aligned} inc &:: Expr \text{ Int} \rightarrow Expr \text{ Int} \\ inc (Lit\ x) &= Lit\ (x + 1) \end{aligned}$$

This solution is incorrect because we also have to write the “boilerplate” code for traversing the entire expression tree, which just leaves the structure intact and recurses into the arguments. Using SYB, we do not have to do that anymore: we signal that the other cases are uninteresting by saying:

$$inc\ x = x$$

Now we have the complete definition of function *inc*: increment the literals and leave the rest untouched. To ensure that this function is applied everywhere in the expression we write:

$$\begin{aligned} increment &:: Data\ a \Rightarrow a \rightarrow a \\ increment &= everywhere\ (mkT\ inc) \end{aligned}$$

This is all we need: the *increment* function increases the value of each literal by one in any Expr. It even works for LinearExprs, or LinearSystems, with no added cost.

We now proceed to explore the internals of SYB to better understand the potential of this approach and the mechanisms involved behind a simple generic function such as *increment*.

7.2 Run-time type representation

Contrary to the approaches to generic programming discussed earlier, SYB does not provide the structure of datatypes to the programmer, but instead offers basic combinators for writing generic programs. At the basis of these combinators is the method *typeOf* of the type class *Typeable*. Instances of this class can be automatically derived by the GHC compiler, and implement a unique representation of a type, enabling run-time type comparison and type-safe casting.

```
class Typeable a where  
  typeOf :: a → TypeRep
```

An instance of *Typeable* only provides a TypeRep (type representation) of itself. The automatically derived instances of this class by GHC are guaranteed to provide a unique representation for each type, which is a necessary condition for the type-safe cast, as we will see later. So, providing an instance is as easy as adding **deriving** *Typeable* at the end of a datatype declaration.

```
data MyDatatype a = MyConstructor a deriving Typeable
```

We will not discuss the internal structure of TypeRep, since a programmer should not define instances manually. However, the built-in derivation of *Typeable* makes SYB somewhat less portable than the previous two libraries we have seen, and makes it impossible to adapt the type representation.

The *Typeable* class is the “back-end” of SYB. The *Data* class can be considered the “front-end”. It is built on top of the *Typeable* class, and adds generic folding, unfolding and reflection capabilities⁸.

```
class Typeable d ⇒ Data d where  
  gfoldl      :: (∀ a b. Data a ⇒ c (a → b) → a → c b)  
              → (∀ g. g → c g)  
              → d  
              → c d  
  gunfold    :: (∀ b r. Data b ⇒ c (b → r) → c r)  
              → (∀ r. r → c r)
```

⁸ The *Data* class has, in fact, many more methods, but they all have default definitions based on these four basic combinators. They are provided as instance methods so that a programmer can define more efficient versions, specialized to the datatype in question.

```

→ Constr
→ c d
toConstr    :: d → Constr
dataTypeOf  :: d → DataType

```

The combinator *gfoldl* is named after the function *foldl* on lists, as it can be considered a “left-associative fold operation for constructor applications,” with *gunfold* being the corresponding unfold. The types of these combinators may be a bit intimidating, and they are probably better understood by looking at specific instances. We will give such instances in the next subsection, since giving an instance of *Data* for a datatype is the way generic functions become available on the datatype.

7.3 Going generic: universe extension

To use the SYB combinators on a particular datatype we have to supply the instances of the datatype for the *Typeable* and the *Data* class. A programmer should not define instances of *Typeable*, but instead rely on the automatically derived instances by the compiler. For *Data*, GHC can also automatically derive instances for a datatype, but we present an instance here to illustrate how SYB works. For example, the instance of *Data* on the *List* datatype is as follows.

```

instance (Typeable a, Data a) => Data (List a) where
  gfoldl k z Nil      = z Nil
  gfoldl k z (Cons h t) = z Cons 'k' h 'k' t
  gunfold k z l      = case constrIndex l of
    1 → z Nil
    2 → k (k (z Cons))

```

Any instance of the *Data* class follows the regular pattern of the above instance: the first argument to *gfoldl* (*k*) can be seen as an application combinator, and the second argument (*z*) as the base case generator. Function *gfoldl* differs from the regular *foldl* in two ways: it is not recursive, and the base case takes a constructor as argument, instead of a base case for just the *Nil*. When we apply *gfoldl* to function application and the identity function, it becomes the identity function itself.

```

gfoldl ($) id x = x

```

We further illustrate the *gfoldl* function with another example.

```

gsize :: Data a => a → Int
gsize = unBox . gfoldl k (λ_ → IntBox 1) where
  k (IntBox h) t = IntBox (h + gsize t)
newtype IntBox x = IntBox { unBox :: Int }

```

Function *gsize* returns the number of constructors that appear in a value of any datatype that is an instance of *Data*. For example, if it is applied to a list containing pairs, it will count both the constructors of the datatype *List*, and of the datatype for pairs. Given the general type of *gfoldl*, we have to use a container type for the result type *Int* and perform additional boxing and unboxing.

Function *gunfold* acts as the dual operation of the *gfoldl*: *gfoldl* is a generic consumer, which consumes a datatype value generically to produce some result, and *gunfold* is a generic producer, which consume a datatype value to produce a datatype value generically. Its definition relies on *constrIndex*, which returns the index of the constructor in the datatype of the argument.

The two other methods of class *Data* which we have not yet mentioned are *toConstr* and *dataTypeOf*. These functions return, as their names suggest, constructor and datatype representations of the term they are applied to. We continue our example of the *Data* instance for the *List* datatype⁹.

```

toConstr Nil          = con1
toConstr (Cons _ _) = con2
dataTypeOf _         = ty
con1 = mkConstr ty "Empty_List" [] Prefix
con2 = mkConstr ty "Cons_List"  [] Prefix
ty    = mkDataType "ModuleNameHere" [con1, con2]

```

The functions *mkConstr* and *mkDataType* are provided by the SYB library as means for building *Constr* and *DataType*, respectively. *mkConstr* build a constructor representation given the constructor's datatype representation, name, list of field labels and fixity. *mkDataType* builds a datatype representation given the datatype's name and list of constructor representations. These two methods together form the basis of SYB's type reflection mechanism, allowing the user to inspect and construct types at runtime.

Exercise 13. Write a suitable instance of the *Data* class for the *Expr* datatype from Section 3. ■

The basic combinators of SYB are mainly used to define other useful combinators. It is mainly these derived combinators that are used by a generic programmer. Functions like *gunfoldl* appear very infrequently in generic programs. In the next subsection we will show many of the derived combinators in SYB.

7.4 Generic functions in SYB

We now proceed to show a few generic functions in the SYB approach. In SYB, as in many other approaches, it is often useful to first identify the type of the generic function, before selecting the most appropriate combinators to implement it.

⁹ Instead of "ModuleNameHere" one should supply the appropriate module name, which is used for unambiguous identification of a datatype.

Types of SYB combinators

Transformations, queries, and builders are some of the important basic combinators of SYB. We discuss the type of each of these.

A transformation transforms an argument value in some way, and returns a value of the same type. It has the following type:

type GenericT = $\forall a. Data\ a \Rightarrow a \rightarrow a$

There is also a monadic variant of transformations, which allows the use of a helper monad in the transformation.

type GenericM m = $\forall a. Data\ a \Rightarrow a \rightarrow m\ a$

A query function processes an input value to collect information, possibly of another type.

type GenericQ r = $\forall a. Data\ a \Rightarrow a \rightarrow r$

A builder produces a value of a particular type.

type GenericB = $\forall a. Data\ a \Rightarrow a$

A builder that has access to an auxiliary computation is called a “reader”.

type GenericR m = $\forall a. Data\ a \Rightarrow m\ a$

The type does not require m to be a monad.

Many functions in the SYB library are suffixed with one of the letters T , M , Q , B , or R to help identify their usage. Examples are the functions mkT , mkQ , mkM , $extB$, $extR$, $extQ$, $gmapT$ and $gmapQ$, some of which are defined in the rest of this section.

Basic examples

Recall the *increment* function with which we started Section 7.1. Its definition uses the higher-order combinators *everywhere* and mkT . The former is a traversal pattern for transformations, applying its argument everywhere it can:

$everywhere :: GenericT \rightarrow GenericT$
 $everywhere\ f = f . gmapT\ (everywhere\ f)$

Function $gmapT$ maps a function only to the immediate subterms of an expression. It is defined using $gfoldl$ as follows:

$gmapT :: Data\ a \Rightarrow (\forall b. Data\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a$
 $gmapT\ f\ x = unID\ (gfoldl\ k\ ID\ x)$
where
 $k\ (ID\ c)\ y = ID\ (c\ (f\ y))$
newtype ID $x = ID\ \{unID :: x\}$

Exercise 14. Function *everywhere* traverses a (multiway) tree. Define

```
everywhere' :: GenericT → GenericT
```

as *everywhere* but traversing in the opposite direction. ■

Function *mkT* lifts a (usually type-specific) function to a function that can be applied to a value of any datatype.

```
mkT :: (Typeable a, Typeable b) ⇒ (b → b) → a → a
```

For example, *mkT* (*sin* :: Float → Float), applies function *sin* if the input value is of type Float, and the identity function to an input value of any other type. The combination of the two functions *everywhere* and *mkT* allows us to lift a type-specific function to a generic function and apply it everywhere in a value.

Proceeding from transformations to queries, we define a function that sums all the integers in an expression.

```
total :: GenericQ Int  
total = everything (+) (0 'mkQ' lit) where  
  lit :: Expr Int → Int  
  lit (Lit x) = x  
  lit x      = 0
```

Queries typically use the *everything* and *mkQ* combinators. Function *everything* applies its second argument everywhere in the argument, and combines results with its first argument. Function *mkQ* lifts a type-specific function of type $a \rightarrow b$, together with a default value of type b , to a generic a query function of type `GenericQ b`. If the input value is of type a , then the type-specific function is applied to obtain a b value, otherwise it returns the default value. To sum the literals in an expression, function *total* combines subresults using the addition operator (+), and it keeps occurrences of literals, whereas all other values are replaced by 0.

Generic maps

Functions such as *increment* and *total* are defined in terms of functions *everywhere*, *everything*, and *mkT*, which in turn are defined in terms of the basic combinators provided by the *Data* and *Typeable* classes. Many generic functions are defined in terms of combinators of the *Data* class directly, as in the examples below. We redefine function *gsize* defined in Section 7.3 using the combinator *gmapQ*, and we define a function *glength*, which determines the number of children of a constructor, also in terms of *gmapQ*.

```
gsize :: Data a ⇒ a → Int  
gsize t = 1 + sum (gmapQ gsize t)  
glength :: GenericQ Int  
glength = length . gmapQ (const ())
```

The combinator $gmapQ$ is one of the mapping combinators in $Data$ class of the SYB library.

$$gmapQ :: (\forall a. Data a \Rightarrow a \rightarrow u) \rightarrow a \rightarrow [u]$$

It is rather different from the regular list map function, in that works on any datatype that is an instance of $Data$, and that it only applies its argument function to the immediate children of the top-level constructor. So for lists, it only applies the argument function to the head of the list and the tail of the list, but it does not recurse into the list. This explains why $gsize$ recursively calls itself in $gmapQ$, while $glength$, which only counts immediate children, does not use recursion.

Exercise 15. Define the function:

$$gdepth :: GenericQ Int$$

which computes the depth of a value of any datatype using $gmapQ$. The depth of a value is the maximum number of constructors on any path to a leaf in the value. For example:

$$\begin{aligned} gdepth [1,2] &\rightsquigarrow 3 \\ gdepth ((Lit 1) + (Lit 2) + (Var "x")) &\rightsquigarrow 4 \end{aligned}$$

■

Exercise 16. Define the function:

$$gwidth :: GenericQ Int$$

which computes the width of a value of any datatype using $gmapQ$. The width of a value is the number of elements that appear at a leaf. For example:

$$\begin{aligned} gwidth () &\rightsquigarrow 1 \\ gwidth (Just 1) &\rightsquigarrow 1 \\ gwidth ((1,2), (1,2)) &\rightsquigarrow 4 \\ gwidth (((1,2),2), (1,2)) &\rightsquigarrow 5 \end{aligned}$$

■

Equality

Defining the generic equality function is a relatively simple task in the libraries we have introduced previously. Defining equality in SYB is not that easy. The reason for this is that the structural representation of datatypes is not exposed directly—in SYB, generic functions are written using combinators like $gfoldl$. To define generic equality we need to generically traverse two values at the same

time, and it is not immediately clear how we can do this if *gfoldl* is our basic traversal combinator.

To implement equality, we need a generic zip-like function that can be used to pair together the children of the two argument values. Recall the type of Haskell's *zipWith* function.

$$\text{zipWith} :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$$

However, we need a generic variant that works not only for lists but for any datatype. For this purpose, SYB provides the *gzipWithQ* combinator.

$$\text{gzipWithQ} :: \text{GenericQ } (\text{GenericQ } c) \rightarrow \text{GenericQ } (\text{GenericQ } [c])$$

The type of *gzipWithQ* is rather intricate, but if we unfold the definition of *GenericQ*, and omit the occurrences of \forall and *Data*, the argument of *gzipWithQ* has type $a \rightarrow b \rightarrow c$. It would take too much space to explain the details of *gzipWithQ*. Defining equality using *gzipWithQ* is easy:

```
geq :: Data a => a -> a -> Bool
geq x y = geq' x y
  where
    geq' :: GenericQ (GenericQ Bool)
    geq' x' y' = (toConstr x' == toConstr y') & and (gzipWithQ geq' x' y')
```

The outer function *eq* is used to constrain the type of the function to the type of equality. Function *geq'* has a more general type since it only uses *gzipWithQ* (besides some functions on Booleans).

7.5 Support for overloading

Suppose we want to implement the generic *show* function. Here is a first attempt using the combinators we have introduced in the previous sections.

```
gshow_s :: Data a => a -> String
gshow_s t = "("
  ++ showConstr (toConstr t)
  ++ concat (gmapQ ((++) " " . gshow_s) t)
  ++
  ")"
```

Function *showConstr* :: *Constr* -> *String* is the only function we have not yet introduced. Its behavior follows immediately from its type: it returns the string representing the name of the constructor. Function *gshow_s* returns the string representation of any input value. However, it does not implement **deriving Show** faithfully: it inserts too many parentheses, and, what's worse, it treats all types in a uniform way, so both lists and strings are shown using the names of the constructors *Cons* and *Nil*.

```
gshows "abc" ~> "((:) (a) ((:) (b) ((:) (c) ([]))))"
```

The problem here is that *gshow_s* is “too generic”: we want its behavior to be non-generic for certain datatypes, such as `String`. To obtain special behavior for a particular type we use the *ext* combinators of the SYB library. Since function *gshow_s* has the type of a generic query, we use the *extQ* combinator:

```
extQ :: (Typeable a, Typeable b) => (a -> q) -> (b -> q) -> a -> q
```

This combinator takes an initial generic query and extends it with the type-specific case given in its second argument. Its implementation relies on type-safe cast:

```
extQ f g a = maybe (f a) g (cast a)
```

Function *cast* relies on the *typeOf* method of the *Typeable* class (the type of which we have introduced in Section 7.2), to guarantee type equality and ultimately uses *unsafeCoerce* to perform the cast.

Using *extQ*, we can now define a better pretty-printer:

```
gshow :: Data a => a -> String
gshow = (λt ->
  "("
  ++ showConstr (toConstr t)
  ++ concat (gmapQ ((++) " ".gshow) t)
  ++ ")")
) 'extQ' (show :: String -> String)
```

Summarizing, the *extQ* combinator (together with its companions *extT*, *extR*, ...) is the mechanism for overloading in the SYB approach.

Exercise 17.

1. Check the behavior of function *gshow* on a value of type `Char`, and redefine it to behave just like the standard Haskell *show*.
2. Check the behavior of *gshow* on standard Haskell lists, and redefine it to behave just like the standard Haskell *show*. Note: since the list datatype has kind $\star \rightarrow \star$, using *extQ* will give problems. This problem is solved in SYB by defining combinators for higher kinds. Have a look at the *ext1Q* combinator.
3. Check the behavior of *gshow* on standard Haskell pairs, and redefine it to behave just like the standard Haskell *show*. Note: now the datatype has kind $\star \rightarrow \star \rightarrow \star$, but *ext2Q* is not defined! Fortunately, you can define it yourself...
4. Make the function more efficient by changing its return type to `ShowS` and using function composition instead of list concatenation.



Exercise 18. Define function `gread :: (Data a) => String -> [(a, String)]`. Decide for yourself how complete you want your solution to be regarding whitespace, infix operators, etc. Note: you don't have to use `gunfold` directly: `fromConstr`, which is itself defined using `gunfold`, can be used instead. ■

7.6 Case study: selections in exercises assistants

One of the extensions to the exercise assistants that we are implementing is that a student may select a subexpression and ask for possible rewrite rules for that subexpression. Before we can present the possible rewrite rules, we want to check if a selected subexpression is *valid*.

Determining the validity of a subexpression may depend on the context. In the general case, a subexpression is valid if it is a typed value that appears in the abstract syntax tree of the original expression. However, in some cases this definition might be too strict. For instance, for arithmetic expressions, the expression `2 + 3` would not be a subexpression of `1 + 2 + 3`, because the plus operator is left-associative, hence only `1 + 2` is a valid subexpression. Therefore we consider a subexpression to be valid if it appears in the original expression modulo associative operators and special cases (such as lists).

Checking whether a subexpression is valid or not can be determined in various ways. It is important to realize that the problem is strongly connected to the concrete syntax of the datatype. The validity of a selection depends on how terms are pretty-printed on the screen. Aspects to consider are fixity and associativity of operators, parentheses, etc. Simply parsing the selection will not give an acceptable solution. For instance, in the expression `1 + 2 * 3`, the selection `1 + 2` parses correctly, but it is not a valid subexpression.

For these reasons, the selections problem depends on parsing and pretty-printing, and the way a datatype is read and shown to the user. Therefore we think that the best way to solve this problem is to devise an extended parser or pretty-printer, which additionally constructs a function that can check the validity of a selection.

However, parsers and pretty-printers for realistic languages are not usually not generic. Typically, operator precedence and fixity are used to reduce the number of parentheses and to make the concrete syntax look more natural. Therefore, parsers and pretty-printers are often hand-written, or instances of a generic function with ad-hoc cases.

For conciseness, we will present only a simple solution to this problem which works for datatypes that are shown with the `gshow` function of the previous section. We will use a state monad transformer with an embedded writer monad. The state monad keeps track of the current position using an `Int`, whereas the writer monad gradually builds a `Map`. Ideally, this would map a selection range (consisting of a pair of `Ints`) to the type of that selection. This is necessary because an expression might contain subexpressions of different types. However, for simplicity we will leave the `Type` as singleton.

```
type Range = (Int, Int)
type Type = ()
```

```

type Selections = Map Range Type
type Pos       = Int
type MyState   = StateT Pos (Writer Selections) ()

```

Using the monad transformer in this way enables us to maintain the position as state while building the output *Map* at the same time, avoiding manual threading of these values.

The top-level function *selections* runs the monads. Within the monads, we first get the current position (*m*). Then we calculate the position at the end of the argument expression (*n*), and add the selection of the complete expression (*m, n*) to the output *Map*. The main worker function *selsConstr* calculates the selections within the children of the top-level node. *selsConstr* defines the general behavior, and through overloading pairs and strings are given ad-hoc behavior.

```

selections :: Data a => a -> Selections
selections t = execWriter (evalStateT (sels' t) 0) where
  sels' :: Data a => a -> MyState
  sels' t = do
    m ← get
    let n = m + length (gshow t)
    tell (M.singleton (m, n) ())
    (selsConstr 'ext2Q' selsPair 'extQ' selsString) t
    put n

```

For the children of the current term we use different functions based on the type. After the children are done we set the current position to the end of this term. This means that the functions that process the children do not need to care about updating the position to reflect finalizing elements (such as a closing bracket, for instance).

Children are dealt with as follows. In case there are no children, the position has to be updated to take into account the opening bracket, the length of the constructor and the closing bracket. If there are children, we recursively apply the worker function to each child. However, the arguments of a constructor are separated by a space, so we have to increment the position in between each child. This is done with *intersperse* (*modify* (+1)). Finally the list of resulting monads is sequenced:

```

selsConstr :: Data a => a -> MyState
selsConstr t = do
  when (nrChildren t > 0) $
    modify (+(2 + length (showConstr (toConstr t))))
    sequence_ $ intersperse (modify (+1)) $ gmapQ sels' t

```

The *nrChildren* function returns the number of children of the argument expression, irrespective of their type.

As with function *gshow*, we need different code to handle some specific types. For pairs and Strings we use the following:

```

selsPair :: (Data a, Data b) => (a, b) -> MyState
selsPair (a, b) = do
    modify (+1)
    sels' a
    modify (+1)
    sels' b

selsString :: String -> MyState
selsString t = return ()

```

The trivial definition of *selsString* ensures that a *String* is not seen as a list of characters.

As mentioned before, the *selections* function presented in this section has been simplified in a number of ways. Possible improvements include support for operator fixity and priority (which change the parentheses), mapping a range to the actual value in the selection, and decoupling from a specific pretty-printer.

Exercise 19. Extend the *selections* function with a specific case for lists. Valid selections within a list are every element and the entire list. Additionally, change *Type* to *Dynamic*. ■

7.7 Making generic functions extensible

The SYB library as described above suffers from a serious drawback: after a generic function is defined, it cannot be extended to have special behavior on a new datatype. We can, as illustrated in Section 7.5 with function *gshow*, define a function with type-specific behavior. But after such function is defined, defining another function to extend the first one with more type-specific behavior is impossible. Suppose we want to extend the *gshow* function with special behavior for a new datatype:

```

data NewDatatype = One String | Two [Int] deriving (Typeable, Data)
gshow' :: Data a => a -> String
gshow' = gshow 'extQ' showNewDatatype where
    showNewDatatype :: NewDatatype -> String
    showNewDatatype (One s) = "String: " ++ s
    showNewDatatype (Two l) = "List: " ++ gshow l

```

Now we have:

```
gshow' (One "a") ~> "String: a"
```

as we expected. However:

```
gshow' (One "a", One "b") ~> "((,) (One \"a\") (One \"b\"))"
```

This example illustrates the problem: as soon as *gshow'* calls *gshow*, the type-specific behavior we just defined is never again taken into account, since *gshow* has no knowledge of the existence of *gshow'*.

To make generic functions in SYB extensible, Lämmel and Peyton Jones [2005] extended the SYB library, lifting generic functions to Haskell's type class system. A generic function like *gsize* is now defined as follows:

```
class Size a where
  gsize :: a → Int
```

The default case is written as an instance of the form:

```
instance ... ⇒ Size a where ...
```

Ad-hoc cases are instances of the form (using lists as an example):

```
instance Size a ⇒ Size [a] where ...
```

This requires overlapping instances, since the default case is more general than any type-specific extension. Fortunately, GHC allows overlapping instances. A problem is that this approach also needs to lift generic combinators like *gmapQ* to a type class, which requires abstraction over type classes. Abstraction over type classes is not supported by GHC. The authors then proceed to describe how to circumvent this by encoding an abstraction using dictionaries. This requires the programmer to write the boilerplate code of the proxy for the dictionary type. We do not discuss this extension and refer the reader to [Lämmel and Peyton Jones, 2005] for further information.

7.8 Variants

Scrap Your Boilerplate Reloaded [Hinze et al., 2006] is a variant of SYB that replaces the combinator based approach of SYB by a tangible representation of the structure of values. The Spine datatype is used to encode the structure of datatypes.

```
data Spine :: * → * where
  Con :: a → Spine a
  (◊) :: Spine (a → b) → Typed a → Spine b
```

where the Typed representation is given by:

```
data Typed a = (◊) { typeOf :: Type a, val :: a }
data Type :: * → * where
  Int :: Type Int
  List :: Type a → Type [a]
  ...
```

This approach represents the structure of datatype values by making the application of a constructor to its arguments explicit. For example, the list $[1, 2]$ can be represented by¹⁰ $\text{Con } (:) \diamond (\text{Int } \hat{\diamond} 1) \diamond (\text{List } \text{Int } \hat{\diamond} [2])$. We can define the

¹⁰ Note the difference between the list constructor $(:)$ and the Typed constructor $(\hat{\diamond})$.

usual SYB combinators such as *gfoldl* on the Spine datatype. Function *gunfold* cannot be implemented in the approach. Scrap Your Boilerplate Revolutions [Hinze and Löh, 2006] solves this problem by introducing the “type spine” and “lifted spine” views. These views allow the definition of not only generic readers such as *gunfold*, but even functions that abstract over type constructors, such as *fmap*, in a natural way. It has recently been found by [Reinke, 2008] and [Kiselyov, 2008] that *fmap* can be defined in standard SYB. However, the definition is rather intricate, and as such we do not present it in these lecture notes.

A disadvantage of these variants is that generic and non-generic universe extension require recompilation of type representations and generic functions. For this reason, these variants cannot be used as a library, and should be considered a design pattern rather than a library. It is possible to make the variants extensible by using a similar approach as discussed in Section 7.7: abstraction over type classes. We refer the reader to [Hinze et al., 2006, Hinze and Löh, 2006] for further information.

8 Comparison of the libraries

In the sections 5, 6, and 7, we introduced three libraries for generic programming in Haskell. There are many other libraries that we exclude for lack of space (see Section 10.1 for a list). The obvious question a Haskell programmer who wants to implement a generic program now asks is: Which library do I use for my project? The answer to this question is, of course, that it depends. In this section, we present an abridged comparison of the three libraries we have seen, focusing mainly on the differences between them. For further study, we refer the reader to a recent, extensive comparison of multiple generic programming libraries and their characteristics [Rodriguez et al., 2008b].

8.1 Differences

There are a few aspects in which the three libraries we have presented differ considerably.

Universe size

What are the datatypes for which generic universe extension is possible? In Section 3, we saw a variety of Haskell datatypes. The more datatypes a library can support, the more useful that library will be. None of the libraries supports existentially quantified datatypes or GADTs. On the other hand, all libraries support all the other datatypes mentioned.

SYB’s automatic derivation does not work for higher-order kinded datatypes, but the programmer can still add the instances manually. Datatypes which are both higher-order kinded and nested are not supported by SYB. Both LIGD and EMGM can support such datatypes, but they cannot be used with EMGM’s representation dispatchers.

Subuniverses

Supporting subuniverses means that it is possible to restrict the use of a generic function to a particular set of datatypes. A library which supports subuniverses flags uses of a generic function on datatypes outside of the subuniverse as compile-time errors.

Of the three libraries, only EMGM supports subuniverses. The set of types to which a generic function can be instantiated is controlled by instance declarations. For example, the EMGM implementation of *map* can only be applied to polymorphic datatypes that have a representation based on *Generic2*. Monomorphic types cannot have such representations.

First-class generic functions

If generic functions are first-class, they can be passed as argument to other generic functions. *gmapQ* (as introduced in Section 7.4) is an example of a function which can only be defined if generic functions are first-class.

In LIGD and SYB, a generic function is a polymorphic Haskell function, so it is a first-class value in Haskell implementations that support rank-n polymorphism.

EMGM supports first-class generic functions but in a rather complicated way. The type class instance for a higher-order generic function needs to track calls to a generic function argument. This makes the definition of *gmapQ* in EMGM significantly more complex than other functions.

Ad-hoc definitions for datatypes

A library supports ad-hoc definitions for datatypes if it can define functions with specific behavior on a particular datatype while the other datatypes are handled generically. Moreover, the use of ad-hoc cases should not require recompilation of existing code (for instance the type representations).

In LIGD, giving ad-hoc cases requires extending the type representation datatype, and hence recompilation of the module containing type representations. This means the library itself must be changed, so we consider LIGD not to support ad-hoc definitions.

In SYB, ad-hoc cases for queries are supported by means of the *mkQ* and *extQ* combinators. Such combinators are also available for other traversals, for example transformations. The only requirement for ad-hoc cases is that the type being case-analyzed should be an instance of the *Typeable* type class. The new instance does not require recompilation of other modules. In EMGM, ad-hoc cases are given as instances of *Rep*, *FRep*, or one of the other representation dispatchers. Recompilation of the library is not required, because ad-hoc cases are given as type class instances.

Extensibility

If a programmer can extend the universe of a generic function in a different module without the need for recompilation, then the approach is extensible.

This is the case for libraries that allow the extension of the generic *show* function with a case for printing lists, for instance. Extensibility is not possible for approaches that do not support ad-hoc cases. For this reason, LIGD is not extensible.

The SYB library supports ad-hoc definitions, but does not support extensible generic functions (as outlined in Section 7.7).

In EMGM, ad-hoc cases are given in instance declarations, which may reside in separate modules; therefore, the library supports extensibility.

Overhead of library use

The overhead of library use can be compared in different ways including automatic generation of representations, number of structure representations, and amount of work to define and instantiate a generic function.

SYB is the only library that offers support for automatic generation of representations. It relies on GHC to generate *Typeable* and *Data* instances for new datatypes. This reduces the amount of work for the programmer.

The number of structure representations is also an important factor of overhead. LIGD and EMGM have two sorts of representations: a representation for kind \star types and representations for type constructors, which are arity-based. The latter consists of a number of arity-specific representations. For example, to write the *map* function we have to use a representation of arity two. Since there are useful generic functions requiring a representation of arity three, this makes a total of four type representations for these libraries: one to represent kind \star types, and three for all useful arities. In SYB, the structure representation is given in a *Data* instance. This instance has two methods which are used for generic consumer and transformer functions (*gfoldl*) and generic producer functions (*gunfold*). Therefore, every datatype needs two representations to be used with SYB functions.

Instantiating a generic function should preferably also be simple. Generic functions require a value representing the type to which they are instantiated. This representation may be explicitly supplied by the programmer or implicitly derived. In approaches that use type classes, representations can be derived, thus making instantiation easier for the user. Such is the case for SYB and EMGM. LIGD uses an explicit type representation, which the user has to supply with every generic function call.

Practical aspects

With practical aspects we mean the availability of a library distribution, quality of documentation, predefined generic functions, etc.

LIGD does not have a distribution online. EMGM recently gained an online status with a website, distribution, and extensive documentation [Utrecht, 2008]. Many generic functions and common datatype representations are provided. SYB is distributed with the GHC compiler. This distribution includes a number of traversal combinators for common generic programming tasks and

Haddock documentation. The GHC compiler supports the automatic generation of *Typeable* and *Data* instances.

Portability

The fewer extensions of the Haskell 98 standard, or of the coming Haskell Prime [Haskell Prime list, 2006] standard, an approach requires, the more portable across different Haskell compilers it is.

Generics for the Masses [Hinze, 2004], as originally introduced, was very portable. EMGM as described is also quite portable, although it does rely on multiparameter type classes to support implicit type representations (currently slated to become part of Haskell Prime). However, we must be careful with the latest change to EMGM to support convenient extensibility (Section 6.6), because it relies on overlapping instances. It is supported by several Haskell compilers, but the implementations of overlapping instances may not necessarily be equivalent.

LIGD, as presented here, relies on GADTs for the type representation. It is not yet clear if GADTs will be included in Haskell Prime. LIGD also requires rank-2 types for the representations of higher-kinded datatypes, but not for other representations or functions. Hence rank-n types are not essential for the LIGD approach.

SYB requires rank-n polymorphism for the type of the *gfoldl* and *gunfold* combinators, *unsafeCoerce* to implement type safe casts and compiler support for deriving *Data* and *Typeable* instances. Hence, it is the least portable of the three libraries.

8.2 Similarities

There are a couple of aspects in which the libraries are similar.

Abstraction over type constructors

Generic functions like *map* or *crush* require abstraction over type constructors to be defined. Type constructors are types which expect a type argument (and therefore have kind $\star \rightarrow \star$), and represent containers of elements. All libraries support the definition of such functions, although the definition in SYB is rather cumbersome.

Separate compilation

Is generic universe extension modular? A library that can instantiate a generic function to a new datatype without recompiling the function definition or the type/structure representation is modular.

All presented libraries are modular. In LIGD, representation types have a constructor to represent the structure of datatypes, namely *RType*. It follows that generic universe extension requires no extension of the representation datatypes

and therefore no recompilation. In EMGM, datatype structure is represented by *rtype*, so a similar argument applies. In SYB, generic universe extension is achieved by defining *Data* and *Typeable* instances for the new datatype, which does not require recompilation of existing code in other modules.

Multiple arguments

Can a generic programming library support a generic function definition that consumes more than one generic argument? Functions such as generic equality require this. The LIGD and EMGM approaches support the definition of generic equality. Furthermore, equality is not more difficult to define than other consumer functions. Equality can also be defined in SYB, but the definition is not as direct as for other functions such as *gshow*. In SYB, the *gfoldl* combinator processes just one argument at a time. For this reason, the definition of generic equality has to perform the traversal of the arguments in two stages using the generic zip introduced in Section 7.4.

Constructor names

All generic programming libraries discussed in these notes provide support for constructor names in their structure representations. These names are used by generic show functions.

Consumers, transformer and producers

LIGD and EMGM can define consumer, transformer, and producer functions. SYB can also define them, but consumers and producers are written using different combinators.

9 Type-indexed datatypes with type families

So far we have been looking at libraries for generic programming and what functions can be expressed in these libraries. A strongly related concept in generic programming is the concept of type-indexed datatypes [Hinze et al., 2002]. Such datatypes are constructed in a generic way from an argument datatype.

For example, a type-indexed datatype is needed when we want to represent datatypes with “holes”. This might be necessary when editing a value: we start editing with a hole, and gradually fill in the details of our object, or parts of an expression might be removed and later filled in. To allow for holes, the original datatype has to be extended with an extra constructor which represents a hole. However, the datatype might not even be accessible (if it is taken from a library, for instance). The best solution is to define a new datatype which takes a datatype as argument, and adds to it a new constructor for holes. This can be done using a type-indexed datatype.

Other examples of type-indexed types are: extending a datatypes with variables, for example for the purpose of rewriting or unification [van Noort et al., 2008], the zipper and variants [McBride, 2001, Hinze et al., 2002, McBride, 2008], the algebra type for folds [Malcolm, 1990b], generic tries [Hinze, 2000b], etc.

Type-indexed datatypes are available in Generic Haskell [Löh, 2004], an extension of Haskell with generic programming constructs. The generic programming libraries available for Haskell do not offer support for defining type-indexed datatypes. However, it is possible to implement them directly in Haskell using type families [Chakravarty et al., 2005a,b, Schrijvers et al., 2008], a recent extension of Haskell implemented in GHC.

Type families are actively developed¹¹ in GHC. They provide a limited version of named functions at the type level. Their relation with vanilla type constructors is comparable to the relation between polymorphic functions and ad-hoc methods of type classes. While polymorphic functions behave the same way for all type instances, class methods can behave differently depending on the class type parameter. For this reason, type families are frequently presented alongside a class declaration. In this case, they are usually called *associated type synonyms*.

Type families exist in two flavors: data families and type synonym families. The former correspond to algebraic datatypes and the latter to type synonyms. Their usefulness for generic programming is two-fold. They can be used to implement type-indexed functions, just like the approaches we have seen in the previous sections. Even more interesting, however, is their ability to implement type-indexed datatypes.

9.1 Generic insertion

We will show the possibilities of type families through an implementation of a generic insertion function. This function takes a list of elements to insert and a value of a datatype extended with holes, possibly containing holes. It returns the same value in which the holes have been filled, in order, with elements from the list. The function fails if there are not enough values in the list to fill all holes.

Before we define the function, we will define the type-indexed datatype for extending a datatype with a hole. As a first attempt, we could try to add a hole by defining a representation datatype as a sum of a unit and the original datatype. For instance, in LIGD we would define:

```
type Ext a = Unit :+: a
```

`Ext (List a)` represents the datatype of lists extended with holes. However, this definition only adds holes at the top-level of a datatype. `Ext (List a)` expands to `Unit :+: (List a)`, which is isomorphic to `Maybe (List a)`. The problem is that we

¹¹ A working implementation is already present in version 6.8.3, with the full implementation expected for version 6.10. See also <http://hackage.haskell.org/trac/ghc/wiki/TypeFunctions>.

have to add holes at each recursive occurrence of a list, and not just at the top level.

To get holes that may appear anywhere in a value, we need to use a different generic view [Holdermans et al., 2006]. With a fixed-point view we can identify the recursive points of a datatype, and therefore define hole extension adequately. The structure constructors for a fixed-point view are shown below:

```

data Unit    r = Unit
data Id      r = Id  {unId :: r}
data K a     r = K a
data Sum f g r = Inl (f r) | Inr (g r)
data Prod f g r = Prod (f r) (g r)

```

Like in the sum of products view (used by LIGD and EMGM), we have a case for Units, Sums and Prods. However, these now take an extra type argument, which is used to encode the recursive argument. Recursive occurrences are represented by the *Id* case, which simply contains the *r* type variable used to encode recursion. Finally, constants (such as *Int* or type variables) are encoded with *K*.

We now define a type class to aggregate regular types that can be viewed as fixed-points. This class defines an associated type synonym *PF a*, which stands for the “pattern functor” of type *a*, and will be the structure representation of the type. We call this class *Regular* since only regular datatypes can be given an instance (recall the definition of *regular* in Section 2.7).

```

class Regular a where
  type PF a :: * -> *
  from      :: a -> PF a a
  to        :: PF a a -> a

```

It also defines the conversion functions which represent the isomorphism between a type *a* and its pattern functor *PF a a*. More specifically, *from* transforms the top-level constructor into a structure value, while leaving the immediate subtrees unchanged. The function *to* performs the transformation in the opposite direction.

Note that the type synonym *PF a* is of kind $* \rightarrow *$. This is because the pattern functor is parametrised over the recursive argument. In our case, *PF a* is applied to *a* itself. This means that we have a shallow representation: at top level we have the structural representation, but at the next level we have the original datatype itself. This representation has the advantage of increased efficiency, since values are only converted to their structural representation when necessary. To fully convert a value, the conversion functions *to* and *from* can be applied recursively.

As an example, we instantiate the standard list datatype on the *Regular* class:

```

instance Regular [a] where
  type PF [a] = Sum Unit (Prod (K a) Id)

```

The pattern functor for lists is a sum, since there are two ways to construct a list. The empty list is associated with the `Unit` value, since the empty list constructor has no arguments. The `(:)` constructor is encoded as a product of the argument `a` (which is a constant) and a recursive occurrence of a list (represented by `ld`).

Now we only have to provide the conversion functions between lists and their structural counterpart:

```

from []                = Inl Unit
from (x : xs)         = Inr (Prod (K x) (Id xs))
to (Inl Unit)         = []
to (Inr (Prod (K x) (Id xs))) = x : xs

```

Instantiating a type to the *Regular* class effectively represents universe extension in this approach, since we are providing the means for generic operations on the datatype.

With this representation in place, we can now properly define the type-indexed datatype which extends a datatype with holes. It is enough to introduce a sum type to encode the choice between the hole case and a value from the original pattern functor:

```
type Ext f = Sum Unit f
```

We still have the same problem as before, namely, `Ext` extends a type with holes on the top-level only, but we also have to allow holes to occur in subterms. To this end, we introduce a type synonym `Hole` that encodes the recursive structure by means of the fixed-point operator `Fix`:

```

newtype Fix f = In{ out :: f (Fix f) }
type Hole f = Fix (Ext (PF f))

```

The `Fix` operator encodes explicit recursion, which allows us to state that the recursive points are `Ext (PF f)` instead of simply `f`, therefore allowing holes to appear at any level of the datatype.

Having a run-time type representation and a mechanism for universe extension, we are ready to define generic functions. We define a monadic traversal function (a variant on the *traverse* function in the *Data.Traversable* module):

```

class TraverseM f where
  traverseM :: Monad m => (a -> m b) -> f a -> m (f b)
instance TraverseM Unit where
  traverseM f Unit = return Unit
instance TraverseM Id where
  traverseM f (Id x) = f x >>= return . Id
instance TraverseM (K a) where
  traverseM _ (K a) = return (K a)
instance (TraverseM f, TraverseM g) => TraverseM (Sum f g) where

```

```

instance (TraverseM f, TraverseM g) => TraverseM (Prod f g) where
traverseM f (Prod x y) = do
  x' ← traverseM f x
  y' ← traverseM f y
  return (Prod x' y')

```

The most interesting case is for the `Id` datatype. Since we are at a recursive point, the function `f` is applied. In all other cases we just recurse through the datatype. Using the monadic traversal function, we can finally define the worker function that inserts values in a value of a datatype extended with holes:

```

insert' :: Insert a => Hole a -> State [a] (Maybe a)
insert' (In (Inl Unit)) = do
  l ← get
  case l of
    [] → return Nothing
    (x : t) → put t >> return (Just x)
insert' (In (Inr x)) = do
  t ← traverseM insert' x
  return (traverseM id t >>= return .to)

```

We keep the list with values to insert as state in a monad. When we encounter a hole (signalled by a left injection), we return the element at the head of the list or fail if there are no more elements. When we encounter a regular value (signalled by a right injection), we use `traverseM` to recursively apply the insertion function to the value. Afterwards, we traverse the value again to propagate the `Maybe` monad to the top level (note that the expression `traverseM id` has type $(Monad\ m, TraverseM\ f) \Rightarrow f\ (m\ a) \rightarrow m\ (f\ a)$). These two traversals could probably be fused into a single one for better efficiency, but the current approach is easy to read and understand.

We define the auxiliary `insert` function which just runs the state monad:

```

insert :: Insert a => [a] -> Hole a -> Maybe a
insert as h = evalState (insert' h) as

```

The `Insert` class is a simple synonym for types that have a pattern functor view and can be traversed:

```

class (Regular a, TraverseM (PF a)) => Insert a

```

This synonym is useful to allow us to get automatic instances from the compiler. As an example, we show the instantiation of the `Expr` datatype (introduced in Section 3):

```

instance Regular (Expr a) where
  type PF (Expr a) = Sum (K String)

```

```

                                (Sum (K a)
                                (Sum (Prod Id Id)
                                (Sum (Prod Id Id)
                                (Sum (Prod Id Id)
                                (Prod Id Id))))))
from (Var s)  = Inl (K s)
from (Lit x)  = Inr (Inl (K x))
from (e1 + e2) = Inr (Inr (Inl (Prod (Id e1) (Id e2))))
from (e1 - e2) = Inr (Inr (Inr (Inl (Prod (Id e1) (Id e2))))))
from (e1 × e2) = Inr (Inr (Inr (Inr (Inl (Prod (Id e1) (Id e2))))))
from (e1 ÷ e2) = Inr (Inr (Inr (Inr (Inr (Prod (Id e1) (Id e2))))))
to (Inl (K s))                                = Var s
to (Inr (Inl (K x)))                          = Lit x
to (Inr (Inr (Inl (Prod (Id e1) (Id e2)))))) = e1 + e2
to (Inr (Inr (Inr (Inl (Prod (Id e1) (Id e2)))))) = e1 - e2
to (Inr (Inr (Inr (Inr (Inl (Prod (Id e1) (Id e2)))))) = e1 × e2
to (Inr (Inr (Inr (Inr (Inr (Prod (Id e1) (Id e2)))))) = e1 ÷ e2
instance Insert (Expr Int)

```

The last line is used to tell the compiler that the Expr datatype is automatically traversable, since it has a pattern functor view. We can now test insertion in expressions:

```

expr :: Hole (Expr Int)
expr = sum hole (sum (lit 2) hole) where
  value x = In (Inr x)
  hole    = In (Inl Unit)
  sum a b = value (Inr (Inr (Inl (Prod (Id a) (Id b)))))
  lit n   = value (Inr (Inl (K n)))
ins1, ins2 :: Maybe (Expr Int)
ins1 = insert [Lit 4, Lit 6] expr
ins2 = insert [Lit 4] expr

```

As expected, $ins1 \rightsquigarrow Just (Lit\ 4 + (Lit\ 2 + Lit\ 6))$ and $ins2 \rightsquigarrow Nothing$.

We do not go into further details in this section. However, the reader is referred to [van Noort et al., 2008] for a full description of the related problem of datatype-generic rewriting using type families, and also for how to represent expressions with holes in a nicer syntax.

This section has shown how to define a type-indexed datatype using type families. Type-indexed datatypes come naturally with generic functions, and the possibility to define type-indexed datatypes in Haskell itself using type families makes generic programming in Haskell even more attractive.

10 Conclusions

These lecture notes serve as an introduction to generic programming in Haskell. We begin with a look at the context of generics and variations on this theme. The term “generics” usually involves some piece of a program parametrised by some other piece. The most basic form is the function, a computation parametrised by values. A more interesting category is genericity by the shape of a datatype. This has been studied extensively in Haskell, because datatypes plays a central role in program development.

We next explore the world of datatypes. From monomorphic types with no abstraction to polymorphic types with universal quantification to existentially quantified types that can simulate dynamically typed values, there is a wide range of possibilities in Haskell. The importance of datatypes has led directly to a number of attempts to develop methods to increase code reuse when using multiple, different types.

In the last decade, many generic programming approaches have resulted in libraries. Language extensions have also been studied, but libraries have been found to be easier to ship, support, and maintain. We cover three representative libraries in detail: LIGD, EMGM, and SYB. LIGD passes a run-time type representation to a generic function. EMGM relies on type classes to represent structure and dispatching on the appropriate representation. SYB builds generic functions with basic traversal combinators. As a related idea, we introduce type-indexed datatypes and an implementation using type families, a recent extension of Haskell.

Having introduced variants of generic programming libraries in Haskell, we can imagine that the reader wants to explore this area further. For that purpose, we provide a collection of references to help in this regard.

Lastly, we speculate on the future of libraries for generic programming. Given what we have seen in this field, where do we think the research and development work will be next? What are the problems we should focus on, and what advances will help us out?

10.1 Further reading

We provide several categories for further reading on topics related to generic programming, libraries, programming languages, and similar concepts or background.

Generic programming libraries in Haskell

Each of these articles describes a particular generic programming library or approach in Haskell.

LIGD	[Cheney and Hinze, 2002]
SYB	[Lämmel and Peyton Jones, 2003]
	[Lämmel and Peyton Jones, 2004]
PolyLib	[Norell and Jansson, 2004a]
EMGM	[Hinze, 2004, 2006]
	[Oliveira et al., 2006]
SYB with Class	[Lämmel and Peyton Jones, 2005]
Spine	[Hinze et al., 2006]
	[Hinze and Löh, 2006]
RepLib	[Weirich, 2006]
Smash your Boilerplate	[Kiselyov, 2006]
Uniplate	[Mitchell and Runciman, 2007]
Generic Programming, Now!	[Hinze and Löh, 2007]

Generic programming in other programming languages

We mention a few references for generic programming using language extensions and in programming languages other than Haskell.

Generic Haskell	[Löh, 2004, Löh et al., 2008]
OCaml	[Yallop, 2007]
ML	[Karvonen, 2007]
Java	[Palsberg and Jay, 1998]
Clean	[Alimarine and Plasmijer, 2002]
Maude	[Clavel et al., 2000]
Relational languages	[Backhouse et al., 1991]
	[Bird and Moor, 1997]
Dependently typed languages	[Pfeifer and Ruess, 1999]
	[Altenkirch and McBride, 2003]
	[Benke et al., 2003]

Comparison of techniques

Here we list some references comparing different techniques of generic programming, whether that be with language extensions, libraries, or between different programming languages.

Approaches in Haskell	[Hinze et al., 2007]
Libraries in Haskell	[Rodriguez et al., 2008b]
	[Rodriguez et al., 2008a]
Language Support	[Garcia et al., 2007]
C++ Concepts and Haskell Type Classes	[Bernardy et al., 2008]

Background

Lastly, we add some sources that explain the background behind generic programming in Haskell. Some of these highlight connections to theorem proving and category theory.

Generic Programs and Proofs	[Hinze, 2000a]
An Introduction to Generic Programming	[Backhouse et al., 1999]
ADTs and Program Transformation	[Malcolm, 1990a]
Law and Order in Algorithmics	[Fokkinga, 1992]
Functional Programming with Morphisms	[Meijer et al., 1991]

10.2 The future of generic programming libraries

There has been a wealth of activity on generic programming in the last decade and on libraries for generic programming in Haskell in the last five years. Generic programming is spreading through the community, and we expect the use of such techniques to increase in the coming years. Generic programming libraries are also getting more mature and more powerful, and the number of examples of generic programs is increasing.

We expect that libraries will replace language extensions such as Generic Haskell—and possibly Generic Clean [Alimarine and Plasmijer, 2002]—since they are more flexible, easier to maintain and distribute, and often equally as powerful. In particular, if the community adopts type families and GADTs as common programming tools, there is no reason to have separate language extensions for generic programming.

Generic programs are useful in many software packages, but we expect that compilers and compiler-like programs will particularly profit from generic programs. However, to be used in compilers, generic programs must not introduce performance penalties. At the moment, GHC’s partial evaluation techniques are not powerful enough to remove the performance penalty caused by transforming values of datatypes to values in type representations, performing the generic functionality, and transforming the result back again to the original datatype. By incorporating techniques for partial evaluation of generic programs [Alimarine and Smetsers, 2004], GHC will remove the performance overhead and make generic programs a viable alternative.

Acknowledgements. This work has been partially funded by the Netherlands Organisation for Scientific Research (NWO), via the Real-life Datatype-Generic programming project, project nr. 612.063.613, and by the Fundação para a Ciência e Tecnologia, via the SFRH/BD/35999/2007 grant.

We are grateful to many people for their comments on these lecture notes. Most notably, Americo Vargas and students of the Generic Programming course at Utrecht University provided feedback on an early version. The attendees at the 2008 Summer School on Advanced Functional Programming provided further reactions.

Bibliography

- Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70431-5.
- Artem Alimarine and Rinus Plasmijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *The 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers*, volume 2312 of LNCS, pages 168–186. Springer, 2002.
- Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of LNCS, pages 16–31. Springer, 2004.
- Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of IFIP, pages 1–20. Kluwer Academic Publishers, 2003.
- Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers B.V., 1991.
- Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In S. Doaitse Swierstra et al., editors, *Advanced Functional Programming*, volume 1608 of LNCS, pages 28–115. Springer, 1999.
- Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.
- Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas Priesnitz. A comparison of C++ concepts and Haskell type classes. In *ACM SIGPLAN Workshop on Generic Programming*. ACM, 2008.
- Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Proceedings of the 4th International Conference on Mathematics of Program Construction, MPC 1998*, volume 1422 of LNCS, pages 52–67. Springer, 1998.
- Richard Bird and Oege de Moor. *Algebra of programming*. Prentice-Hall, 1997. ISBN 0-13-507245-X.
- Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, IFCP 2005*, pages 241–253, 2005a.
- Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 1–13, 2005b.

- James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, Haskell 2002*, pages 90–104. ACM, 2002.
- Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP 2000*, pages 268–279. ACM, 2000.
- Manuel Clavel, Francisco Durán, and Narciso Martí-Oliet. Polytypic programming in Maude. *Electronic Notes in Theoretical Computer Science*, 36:339–360, 2000.
- Chris Dornan, Isaac Jones, and Simon Marlow. Alex User Guide, 2003. URL <http://www.haskell.org/alex>.
- Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 1992.
- Ira R. Forman and Scott H. Danforth. *Putting metaclasses to work: a new dimension in object-oriented programming*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999. ISBN 0-201-43305-2.
- Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, 2007.
- Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719, pages 1–71. Springer, 2007.
- Aleksey Gurtovoy and David Abrahams. The Boost C++ metaprogramming library, 2002. URL http://www.cs.ualberta.ca/~graphics/software/boost/libs/impl/doc/paper/impl_paper.pdf.
- The Haskell Prime list. Haskell prime, 2006. Wiki page at <http://hackage.haskell.org/trac/haskell-prime>.
- Ralf Hinze. Generics for the masses. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 236–243. ACM, 2004.
- Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.
- Ralf Hinze. Generic programs and proofs. *Bonn University, Habilitation*, 2000a.
- Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000b.
- Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In *Generic Programming*, volume 2793 of LNCS, pages 57–96. Springer, 2003a.
- Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming*, volume 2793 of LNCS, pages 1–56. Springer, 2003b.
- Ralf Hinze and Andres Löb. “Scrap Your Boilerplate” revolutions. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC 2006*, volume 4014 of LNCS, pages 180–208. Springer, 2006.
- Ralf Hinze and Andres Löb. Generic programming, now! In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, LNCS, pages 150–208. Springer, 2007.

- Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th International Conference on Mathematics of Program Construction, MPC 2002*, volume 2386 of *LNCS*, pages 148–174. Springer, 2002.
- Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. “Scrap Your Boilerplate” reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, volume 3945 of *LNCS*. Springer, 2006.
- Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Generic Programming, Advanced Lectures*, volume 4719 of *LNCS*. Springer, 2007.
- Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC 2006*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006.
- Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.
- Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. Libraries for generic programming in Haskell. Technical Report UU-CS-2008-025, Department of Information and Computing Sciences, Utrecht University, 2008.
- Vesa A.J. Karvonen. Generics for the working ML’er. In *Proceedings of the 2007 Workshop on ML, ML 2007*, pages 71–82. ACM, 2007.
- Oleg Kiselyov. Smash your boilerplate without class and typeable. <http://article.gmane.org/gmane.comp.lang.haskell.general/14086>, 2006.
- Oleg Kiselyov. Compositional gmap in SYB1. <http://www.haskell.org/pipermail/generics/2008-July/000362.html>, 2008.
- Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation, TLDI 2003*, pages 26–37. ACM, 2003.
- Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 244–255. ACM, 2004.
- Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 204–215. ACM, 2005.
- John Launchbury and Simon Peyton Jones. Lazy functional state threads. *SIGPLAN Conference on Programming Language Design and Implementation*, 29(6): 24–35, 1994.
- Josje Lodder, Johan Jeuring, and Harrie Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.
- Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
- Andres Löh and Ralf Hinze. Open data types and open functions. In Michael Maher, editor, *Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming, PPDP 2006*, pages 133–144. ACM, 2006.

- Andres Löh, Johan Jeuring, Thomas van Noort, Alexey Rodriguez, Dave Clarke, Ralf Hinze, and Jan de Wit. The Generic Haskell user's guide, version 1.80—Emerald release. Technical Report UU-CS-2008-011, Department of Information and Computing Sciences, Utrecht University, 2008.
- Grant Malcolm. *Algebraic data types and program transformation*. PhD thesis, Department of Computing Science, Groningen University, 1990a.
- Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990b.
- Simon Marlow and Andy Gill. Happy User Guide, 1997. URL <http://www.haskell.org/happy>.
- Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, pages 287–295. ACM, 2008.
- Conor McBride. The derivative of a regular type is its type of one-hole contexts. strictlypositive.org/diff.pdf, 2001.
- Lambert Meertens. Calculate polytypically! In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP 1996*, pages 1–16. Springer, 1996.
- Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. *Proceedings 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA 1991*, 523:124–144, 1991.
- Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the 2007 ACM SIGPLAN workshop on Haskell, Haskell 2007*. ACM, 2007.
- Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In Phil Trinder et al., editors, *Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL 2003*, volume 3145 of LNCS, pages 168–184. Springer, 2004a.
- Ulf Norell and Patrik Jansson. Prototyping generic programming in Template Haskell. In Dexter Kozen, editor, *Proceedings of the 7th International Conference on Mathematics of Program Construction, MPC 2004*, volume 3125 of LNCS, pages 314–333. Springer, 2004b.
- Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006*, volume 7, pages 199–216, 2006.
- Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd IEEE Conference on International Computer Software and Applications, COMPSAC 1998*, pages 9–15, 1998.
- Harrie Passier and Johan Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.

- Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, England, 2003.
- Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1993*, pages 71–84. ACM, 1993.
- Holger Pfeifer and Harald Ruess. Polytypic proof construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings 12th International Conference on Theorem Proving in Higher Order Logics*, number 1690 in Lecture Notes in Computer Science, pages 55–72. Springer, 1999.
- Claus Reinke. Traversable functor data, or: X marks the spot. <http://www.haskell.org/pipermail/generics/2008-June/000343.html>, 2008.
- Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. Technical report, Utrecht University, 2008a.
- Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Haskell Symposium 2008*, 2008b.
- Tom Schrijvers, Simon Peyton Jones, Manuel M. T. Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP 2008*, 2008.
- Tim Sheard. Using MetaML: A staged programming language. *Revised Lectures of the Third International School on Advanced Functional Programming*, 1999.
- Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 2002*, pages 1–16. ACM, 2002.
- Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
- Universiteit Utrecht. EMGM, 2008. URL <http://www.cs.uu.nl/wiki/GenericProgramming/EMGM>.
- Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *ACM SIGPLAN Workshop on Generic Programming*, 2008.
- Philip Wadler. Theorems for free! In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA 1989*, pages 347–359. ACM, 1989.
- Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and Functional Programming, LFP 1990*, pages 61–78. ACM, 1990.
- Philip Wadler. The essence of functional programming. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1992*, pages 1–14, 1992.
- Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.
- Stephanie Weirich. RepLib: a library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, Haskell 2006*, pages 1–12. ACM, 2006.

Jeremy Yallop. Practical generic programming in OCaml. In *Proceedings of the 2007 workshop on Workshop on ML, ML 2007*, pages 83–94. ACM, 2007.