Libraries for Generic Programming in Haskell

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

Universiteit Utrecht, 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

In the development of software, *structuring data* plays an important role. 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, following common programming patterns. Examples of such patterns are:

- in a possibly large value of a complicated datatype (for example for representing the structure of a company), applying a given action at all occurrences of a particular constructor (e.g., adding or updating zip codes at all occurrences of street addresses) while leaving the rest of the value unchanged;
- serializing a value of a datatype, or comparing two values of a datatype for equality, functionality that depends only on the *structure* of the datatype;
- adapting data access functions after a datatype has changed, something that often involves modifying large amounts of existing code.

Generic programming addresses these high-level programming patterns. We also use the term datatype-generic programming [Gibbons, 2007] 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.

Often an instance of a datatype-generic program on a particular datatype is obtained by implementing the instance by hand, a boring and error-prone task, which reduces

P. Koopman and D. Swierstra (Eds.): AFP 2008, LNCS 5832, pp. 165–229, 2009.

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. 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 loss of expressiveness compared with a generic programming language such as Generic Haskell is limited.

These lecture notes introduce generic programming in Haskell using generic programming libraries. We introduce several characteristic generic programming libraries, and we show how to write generic programs using these libraries. 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 starts 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 since we focus on generics in these lecture notes. 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 conclude with some suggested reading and some thoughts about the future of generic programming libraries.

2 Generic Programming in Context (and in Haskell)

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.

Gibbons [2007] lists seven categories of generic programming. In this section, we revisit these with an additional twist: we look at how each may be implemented in 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, etc.), and it is a fundamental element in mathematics. While a function is not often considered under the definition of "generic," it is perfectly reasonable to model other forms of genericity as functions. The generalization is given by the function $g(x)$ in which a generic component g is parametrised by some entity x. Instantiation of the generic component is then analogous to application of a function.

Functions come naturally in Haskell. For example, here is function that takes two boolean values as arguments and determines their basic equality.

$$eq_{Bool} :: \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$$
$$eq_{Bool}\ x\ y = (not\ x \wedge not\ y) \vee (x \wedge y)$$

We take this opportunity to introduce a few themes used in these lecture notes. We use a stylized form of Haskell that is not necessarily what one would type into a file. Here, we add a subscript, which can simply be typed, and use the symbols $(\wedge)$ and $(\vee)$, which translate directly to the standard operators `(&&)` and `(||)`.

As much as possible, we attempt to use the same functions for our examples, so that the similarities or differences are more evident. Equality is one such running example used throughout the text.

2.2 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.

Suppose we have functions defining the logical conjunction and disjunction of boolean values.

$$
\begin{aligned}
&\mathit{and} :: \mathsf{List\ Bool} \to \mathsf{Bool} \\
&\mathit{and}\ \mathit{Nil} &&= \mathit{True} \\
&\mathit{and}\ (\mathit{Cons}\ p\ ps) &&= p \wedge \mathit{and}\ ps \\
\\
&\mathit{or} :: \mathsf{List\ Bool} \to \mathsf{Bool} \\
&\mathit{or}\ \mathit{Nil} &&= \mathit{False} \\
&\mathit{or}\ (\mathit{Cons}\ p\ ps) &&= p \vee \mathit{or}\ ps
\end{aligned}
$$

These two functions exhibit the same recursive pattern. To abstract from this pattern, we abstract over the differences between *and* and *or* using a higher-order function. The pattern that is extracted is known in the Haskell standard library as *foldr* ("fold from the right").

$$
\begin{aligned}
&\mathit{foldr} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{List\ a} \to \mathsf{b} \\
&\mathit{foldr}\ f\ n\ \mathit{Nil} &&= n \\
&\mathit{foldr}\ f\ n\ (\mathit{Cons}\ x\ xs) &&= f\ x\ (\mathit{foldr}\ f\ n\ xs)
\end{aligned}
$$

The *foldr* captures the essence of the recursion in the *and* and *or* by accepting parameters for the *Cons* and *Nil* cases. We can then redefine *and* and *or* in simpler terms using *foldr*.

$$
\begin{aligned}
&\mathit{and} &&= \mathit{foldr}\ (\wedge)\ \mathit{True} \\
&\mathit{or} &&= \mathit{foldr}\ (\vee)\ \mathit{False}
\end{aligned}
$$

2.3 Type

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

Haskell[1] has excellent support for parametrised datatypes and polymorphic functions. The canonical example of the former is List:

$$\mathbf{data}\ \mathsf{List\ a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a})$$

List a is a datatype parametrised by some type a. It is one of a class of datatypes often called "container" types because they provide structure for storing elements of some arbitrary type.

A typical polymorphic function is *length*:

$$
\begin{aligned}
&\mathit{length} :: \mathsf{List\ a} \to \mathsf{Int} \\
&\mathit{length}\ \mathit{Nil} &&= 0 \\
&\mathit{length}\ (\mathit{Cons}\ x\ xs) &&= 1 + \mathit{length}\ xs
\end{aligned}
$$

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

The function *length* can be applied to a value of type List a, and it will return the number of a elements.

An important point to note about parametrised datatypes and polymorphic functions is that they have no knowledge of the parameter. We later discuss other forms of genericity that support increased amounts of information about the parameter.

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 (in Section 2.7). Instead, we use *interface* as the set of requirements needed for instantiation.

Haskell supports a form of interface using type classes and constraints on functions [Wadler and Blott, 1989], which we illustrate with equality. Equality is not restricted to a single type, and in fact, many different datatypes support equality. But unlike the polymorphic *length*, equality on lists for example requires inspection of the elements. The code below defines the class of types that support equality ($(\texttt{==})$) and inequality ($(\not\texttt{==})$).

```
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 some (optional) default implementations. For a datatype such as List a to support the operations in the class *Eq*, we create an instance of it.

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

Notice that our instance for List a requires an instance for a. This is indicated by the context $(Eq\ \texttt{a}) \Rightarrow$.

Methods in type classes and functions that use these methods require a context as well. Consider the observable type of the equality method.

```
(==) :: (Eq a) ⇒ a → a → Bool
```

This function specifies that the type parameter a must be an instance of the *Eq* class. In other words, the type substituted for a must implement the interface specified by *Eq*. This approach is called *ad-hoc polymorphism*. Relatedly, since each recursive call in the definition of the function $(\texttt{==})$ may have a different type, we also describe the function as having *polymorphic recursion*.

2.5 Property

Gibbons expands the concept of generic programming to include the specifications of programs. These *properties* are "generic" in the sense that they may hold for multiple implementations. Properties may be informally or formally defined, and depending on the language or tool support, they may be encoded into a program, used as part of the testing process, or simply appear as text.

A simple example of a property can be found by looking at the methods of the *Eq* type class defined in Section 2.4: a programmer with a classical view on logic would expect that $x == y \equiv not\ (x \neq y)$. This property should hold for all instances of *Eq* to ensure that an instance only needs to define one or the other. However, Haskell's type system provides no guarantee of this. The functions $(==)$ and $(\neq)$ are provided as separate methods to allow for the definition of either (for simplicity) or both (for optimization), and the compiler cannot verify the above relationship. This informal specification relies on programmers implementing the instances of *Eq* such that the property holds.

There are other examples of properties, such as the well-known monad laws [Wadler, 1990], but many of them cannot be implemented directly in Haskell. It is possible, however, to look at a property as an extension of an interface from Section 2.4 if we allow for evaluation in the interface. This can be done in a language with a more expressive type system such as Coq [Bertot and Castéran, 2004] or Agda [Norell, 2007].

2.6 Program Representation

There are numerous techniques in which one program is parametrised by the representation of another program (or its own). This area includes:

- 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].
- Templates in C++ [Alexandrescu, 2001] and multi-stage programming [Taha, 1999] are other techniques.

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

Partly inspired by C++ templates and the multi-stage programming language MetaML [Sheard, 1999], Template Haskell provides a metaprogramming extension to Haskell 98 [Sheard and Peyton Jones, 2002].

We introduce the concept with an example of writing selection functions for tuples of different arities. The standard library provides $fst :: (\mathsf{a}, \mathsf{b}) \to \mathsf{a}$ and $snd :: (\mathsf{a}, \mathsf{b}) \to \mathsf{b}$ since pairs are the most common form of tuples, but for triples, quadruples, etc., we need

to write a new function each time. In other words, we want to automatically generate functions such as these:

$$\begin{aligned} fst3 &= \lambda(x, \_, \_) \quad \to x \\ snd4 &= \lambda(\_, x, \_, \_) \to x \end{aligned}$$

Using Template Haskell, we can write:

$$\begin{aligned} fst3 &= \$\,(sel\ 1\ 3) \\ snd4 &= \$\,(sel\ 2\ 4) \end{aligned}$$

This demonstrates the use of the "splice" syntax, $\$(...)$, to evaluate the enclosed expression 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 implementation[2]:

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

Function *sel* creates an abstract syntax recipe of the form $\lambda(a_1, a_2, ..., ai, ..., an) \to ai$ with a lambda expression (*lamE*), a tuple pattern (*tupP*), variable patterns (*varP*), and a variable expression (*varE*).

Template Haskell is type-safe, and a well-typed program will not "go wrong" at runtime [Milner, 1978]. Initially, splice code is type-checked before compilation, and then the entire program is also type-checked after splice insertion. Compiling the example above may fail for reasons such as the function *sel* not type-checking or the generated code for $\$(sel\ i\ n)$ not type-checking.

Template Haskell has 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].

2.7 Shape

Genericity by shape is, in fact, the focus of these notes. The shape parameter refers to the shape or structure of data. Broadly speaking, if all data has a common underlying set of structural elements, we can write functions that work with those elements. Thus, such functions abstract over any values that can be described by the same shape.

We return again to the example of equality. So far, we have seen two different implementations, one for Bool and one for List, while in fact equality can be defined once generically for many datatypes. In Haskell this generic definition is used when a datatype is annotated with **deriving** *Eq*, but to give you a taste of how this might work in a library, let us look at the shape of some types.

[2] 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.

Intuitively, we can visualize the structural elements by reviewing the syntax for the declaration of the List datatype.

$$\textbf{data}\ \mathsf{List\ a} = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a})$$

First, a List value is either a *Nil* or a *Cons*. This choice between constructors is called a *sum* and denoted by the $+$ symbol in our visualization. Second, each constructor is applied to zero or more arguments. The *Nil* constructor takes no parameters and has the special designator of *unit* represented by $\mathbb{1}$. *Cons*, on the other hand, is applied to two arguments. We use a *product*, indicated by the symbol $\times$, in this case. The representation for List as a whole appears as follows:

$$\textbf{type}\ \mathsf{List}^{\circ}\ \mathsf{a} = \mathbb{1} + (\mathsf{a} \times \mathsf{List\ a})$$

We have now stripped the datatype definition to some basic syntactic elements. Not only can these elements describe the simple List datatype, they also support more complex examples:

$$\textbf{data}\ \mathsf{Map\ k\ a} = \mathit{Tip} \mid \mathit{Bin}\ \mathsf{Int\ k\ a}\ (\mathsf{Map\ k\ a})\ (\mathsf{Map\ k\ a})$$
$$\textbf{type}\ \mathsf{Map}^{\circ}\ \mathsf{k\ a} = \mathbb{1} + (\mathsf{Int} \times (\mathsf{k} \times (\mathsf{a} \times (\mathsf{Map\ k\ a} \times \mathsf{Map\ k\ a}))))$$

The Map datatype from the standard libraries introduces a few new aspects of our syntax. Namely, we can reference other types by name (Int), and if a constructor has more than two arguments (*Bin*), it is represented using a right-associative, nested, product. Furthermore, we reuse the type Map itself in the representation for Map.

The *sum of products* view described above can be used to inductively define functions. We describe the specifics of this for each library in more detail, but for a taste of this process, we define a function in Generic Haskell [Löh, 2004], a language that extends Haskell with syntactic support for datatype-generic programming. Here is equality defined as a generic function:

$$\begin{array}{lllll}
eq\{\!|\mathsf{a} :: \star|\!\}\ ::\ (eq\{\!|\mathsf{a}|\!\}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} & & & & \\
eq\{\!|\mathsf{Int}|\!\} & x & y & = eq_{Int}\ x\ y \\
eq\{\!|\mathsf{Char}|\!\} & c & d & = eq_{Char}\ c\ d \\
eq\{\!|\mathsf{Unit}|\!\} & \mathit{Unit} & \mathit{Unit} & = \mathit{True} \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inl}\ x) & (\mathit{Inl}\ y) & = eq\{\!|\mathsf{a}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & (\mathit{Inr}\ x) & (\mathit{Inr}\ y) & = eq\{\!|\mathsf{b}|\!\}\ x\ y \\
eq\{\!|\mathsf{a} + \mathsf{b}|\!\} & _ & _ & = \mathit{False} \\
eq\{\!|\mathsf{a} \times \mathsf{b}|\!\} & (x_1 \times y_1) & (x_2 \times y_2) & = eq\{\!|\mathsf{a}|\!\}\ x_1\ x_2 \wedge eq\{\!|\mathsf{b}|\!\}\ y_1\ y_2
\end{array}$$

Notice how $eq\{\!|\mathsf{a} :: \star|\!\}$ uses pattern matching on the same structural elements introduced above, which are now types enclosed in $\{\!|\ |\!\}$, to perform case analysis. Looking at each case, we see that the type parameter (e.g. $\mathsf{a} \times \mathsf{b}$) enables the expansion of the value-level structure of the arguments (e.g. $x_1 \times y_1$), thus permitting us to write a separate test of equality specific to each element (e.g. $eq\{\!|\mathsf{a}|\!\}\ x_1\ x_2 \wedge eq\{\!|\mathsf{b}|\!\}\ y_1\ y_2$). We explore these ideas further in the discussion on the libraries LIGD (Section 5) and EMGM (Section 6). For more information on defining generic functions in Generic Haskell,

see Löh [2004] and Hinze and Jeuring [2003a,b]. Also note that Generic Haskell is not the only language extension to datatype-generic programming. A comparison of approaches can be found in Hinze et al. [2007].

There are a number of 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 that we describe in relation to the SYB library (Section 7). For a more in-depth study of generic views, refer to [Holdermans et al., 2006].

In this section, we introduced a variety of techniques that fall under the heading of generic programming; however, this is assuredly not a complete list. For example, research into types that are parametrised by values, often called dependent types, may be also considered "generic." Instead of a thorough description, however, this background should make clear where these lecture notes fit in the broader context of generic programming.

In the next section, we provide more background on the fundamental component of datatype-generic programming: the datatype.

3 The World of Haskell Datatypes

Datatypes play a central role 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 section introduces many of these variations of datatypes by example, and is an updated version of a similar section in Hinze and Jeuring [2003b]. 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:

$$\textbf{data}\ \mathsf{List}_B = \mathit{Nil}_B \mid \mathit{Cons}_B\ \mathsf{Bool}\ \mathsf{List}_B$$

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 $\star$. Other examples of datatypes of kind $\star$ are Int, Char, etc.

Here is the equality function on this datatype:

$$
\begin{array}{lll}
eq_{List_B} :: \mathsf{List}_B \to \mathsf{List}_B \to \mathsf{Bool} & & \\
eq_{List_B}\ Nil_B & Nil_B & = True \\
eq_{List_B}\ (Cons_B\ b_1\ l_1)\ (Cons_B\ b_2\ l_2) & & = eq_{Bool}\ b_1\ b_2 \wedge eq_{List_B}\ l_1\ l_2 \\
eq_{List_B}\ \_ & \_ & = False
\end{array}
$$

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 Parametric Polymorphic Datatypes

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

$$\mathbf{data}\ \mathsf{List\ a} = Nil \mid Cons\ \mathsf{a}\ (\mathsf{List\ a})$$

Compared with List_B, the List a datatype has a different structure: the kind of List is $\star \to \star$. Kinds classify types, just as types classify values. A kind can either be $\star$ (base kind) or $\kappa \to \nu$, where κ and ν are kinds. In Haskell, only the base kind is inhabited, which means there are only values of types of kind $\star$. Since List takes a base type as argument, it has the functional kind $\star \to \star$. The type variable a must be a base type since it appears as a value (as first argument to the *Cons* constructor). In this way, a type of functional kind (such as List) can be fully-applied to create a type of base kind (such as List Int).

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

$$
\begin{array}{lll}
eq_{List} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{List\ a} \to \mathsf{List\ a} \to \mathsf{Bool} & & \\
eq_{List}\ eq_a\ Nil & Nil & = True \\
eq_{List}\ eq_a\ (Cons\ x_1\ l_1)\ (Cons\ x_2\ l_2) & & = eq_a\ x_1\ x_2 \wedge eq_{List}\ eq_a\ l_1\ l_2 \\
eq_{List}\ \_\ \_ & \_ & = False
\end{array}
$$

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 $(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})$ as parameter[3].

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.

[3] Using Haskell's type classes, this would correspond to replacing the type of the first argument in the type of eq_{List} by an $Eq\ \mathsf{a} \Rightarrow$ constraint. The class constraint is later transformed by the compiler into an additional argument of type $(\mathsf{a} \to \mathsf{a} \to \mathsf{Bool})$ to the function.

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

For simplicity, we assume linear expressions are values of a datatype for arithmetic expressions, 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
```

The equality function eq_{Expr} for LinearSystem is straightforward and omitted.

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:

```
eq_Tree :: (a → a → Bool) → Tree a → Tree a → Bool
eq_Tree eq_a Empty        Empty        = True
eq_Tree eq_a (Node a_1 f_1) (Node a_2 f_2) = eq_a a_1 a_2 ∧ eq_Forest eq_a f_1 f_2
eq_Tree _    _            _            = False


eq_Forest :: (a → a → Bool) → Forest a → Forest a → Bool
eq_Forest eq_a Nil          Nil          = True
eq_Forest eq_a (Cons t_1 f_1) (Cons t_2 f_2) = eq_Tree eq_a t_1 t_2 ∧ eq_Forest eq_a f_1 f_2
eq_Forest _    _            _            = False
```

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

A datatype uses higher-order kinds if it is parametrized over a variable of functional kind. All the parametric datatypes we've seen previously took parameters of kind $\star$. Consider the following datatype, which represents a subset of logic expressions.

$$\begin{aligned}\textbf{data}\ \mathsf{Logic_s} &= \mathit{Lit}\ \mathsf{Bool}\\ &\mid \mathit{Not}\ \mathsf{Logic_s}\\ &\mid \mathit{Or}\ \mathsf{Logic_s}\ \mathsf{Logic_s}\end{aligned}$$

Suppose we now want to use the fact that disjunction is associative. For this, we can choose to encode sequences of disjunctions by means of a list. We represent our $\mathsf{Logic_s}$ datatype as:

$$\begin{aligned}\textbf{data}\ \mathsf{Logic_L} &= \mathit{Lit}\ \ \mathsf{Bool}\\ &\mid \mathit{Not}\ \mathsf{Logic_L}\\ &\mid \mathit{Or}\ \ (\mathsf{List}\ \mathsf{Logic_L})\end{aligned}$$

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

$$\begin{aligned}\textbf{data}\ \mathsf{Logic_F}\ \mathsf{f} &= \mathit{Lit}\ \ \mathsf{Bool}\\ &\mid \mathit{Not}\ (\mathsf{Logic_F}\ \mathsf{f})\\ &\mid \mathit{Or}\ \ (\mathsf{f}\ (\mathsf{Logic_F}\ \mathsf{f}))\end{aligned}$$

We have introduced a type variable, and so $\mathsf{Logic_F}$ does not have kind $\star$ as $\mathsf{Logic_L}$. However, its kind is also not $\star \to \star$, as we have seen previously in, for instance, the List datatype, because the type argument that $\mathsf{Logic_F}$ expects is not a base type, but a "type transformer". We can see in the *Or* constructor that f is applied to an argument. The kind of $\mathsf{Logic_F}$ is thus: $(\star \to \star) \to \star$. This datatype is a higher-order kinded datatype.

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

$$\textbf{type}\ \mathsf{Logic'_L} = \mathsf{Logic_F}\ \mathsf{List}$$

Modulo undefined values, $\mathsf{Logic'_L}$ is isomorphic to $\mathsf{Logic_L}$. The type argument of $\mathsf{Logic_F}$ describes which "container" will be used for the elements of the *Or* case.

Defining equality for the $\mathsf{Logic'_L}$ datatype is simple:

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

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

The $\mathsf{Logic_F}$ type requires a somewhat more complicated equality function.

$$
\begin{aligned}
&eq_{Logic_F} :: (\ (\mathsf{Logic_F\ f} \to \mathsf{Logic_F\ f} \to \mathsf{Bool}) \to \\
&\qquad \mathsf{f\ (Logic_F\ f)} \to \mathsf{f\ (Logic_F\ f)} \to \mathsf{Bool}) \to \\
&\quad \mathsf{Logic_F\ f} \to \mathsf{Logic_F\ f} \to \mathsf{Bool} \\
&eq_{Logic_F}\ eq_f\ (Lit\ x_1)\ \ (Lit\ x_2)\ \ = eq_{Bool}\ x_1\ x_2 \\
&eq_{Logic_F}\ eq_f\ (Not\ x_1)\ (Not\ x_2) = eq_{Logic_F}\ eq_f\ x_1\ x_2 \\
&eq_{Logic_F}\ eq_f\ (Or\ x_1)\ \ (Or\ x_2)\ \ = eq_f\ (eq_{Logic_F}\ eq_f)\ x_1\ x_2 \\
&eq_{Logic_F}\ \_\ \_\qquad\quad \_\qquad\quad = False
\end{aligned}
$$

The complexity comes from the need for a higher-order function that itself contains a higher-order function. The function eq_f provides equality on the abstracted container type f, and it needs an equality for its element type $\mathsf{Logic_F\ f}$.

We can specialize this to equality on $\mathsf{Logic_F\ List}$ as follows:

$$
\begin{aligned}
&eq_{Logic_{F,List}} :: \mathsf{Logic_F\ List} \to \mathsf{Logic_F\ List} \to \mathsf{Bool} \\
&eq_{Logic_{F,List}} = eq_{Logic_F}\ (\lambda f\ l_1\ l_2 \to and\ (zipWith\ f\ l_1\ l_2))
\end{aligned}
$$

3.5 Nested Datatypes

A *regular* data type 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.

$$\textbf{data}\ \mathsf{Perfect\ a} = Leaf\ \mathsf{a} \mid Node\ (\mathsf{Perfect\ (a, a)})$$

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:

$$
\begin{aligned}
&eq_{Perfect} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to \mathsf{Perfect\ a} \to \mathsf{Perfect\ a} \to \mathsf{Bool} \\
&eq_{Perfect}\ eq_a\ (Leaf\ x_1)\ \ (Leaf\ x_2)\ \ = eq_a\ x_1\ x_2 \\
&eq_{Perfect}\ eq_a\ (Node\ x_1)\ (Node\ x_2) = eq_{Perfect}\ (eq_{Pair}\ eq_a)\ x_1\ x_2 \\
&eq_{Perfect}\ \_\ \_\qquad\quad \_\qquad\quad = False
\end{aligned}
$$

$$
\begin{aligned}
&eq_{Pair} :: (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{a, a}) \to (\mathsf{a, a}) \to \mathsf{Bool} \\
&eq_{Pair}\ eq_a\ (x_1, x_2)\ (y_1, y_2) = eq_a\ x_1\ x_2 \wedge eq_a\ y_1\ y_2
\end{aligned}
$$

This definition is again very similar to the equality on datatypes we have introduced before. In our case, the container type is the pair of two values of the same type, so in the *Node* case we use equality on this type (eq_{Pair}).

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 → List a → 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:

$$\textbf{data}\ \mathsf{Dynamic} = \forall \mathsf{a}\,.\,\mathit{Dyn}\ (\mathsf{Rep}\ \mathsf{a})\ \mathsf{a}$$

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 ∀ symbol, the type variable a is said to be existentially quantified because it is only available inside the constructor—Dynamic has kind ⋆. 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.

Since an existentially quantified datatype may hide the type of some of its components, the definition of equality may be problematic. If we cannot inspect a component, we cannot compare it. Conversely, we can only compare two values of an existentially quantified datatype 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, but not the values themselves. Therefore equality can only be defined as the equality of the visible components of the existentially quantified 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 $\mathsf{Logic_s}$ and Expr shown before in a datatype for statements:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Stat\ a}\ \textbf{where} \\
\quad \mathit{Val}\ \ :: \mathsf{Expr\ Int} \rightarrow \mathsf{Stat\ (Expr\ Int)} \\
\quad \mathit{Term}\ :: \mathsf{Logic_s} \rightarrow \mathsf{Stat\ Logic_s} \\
\quad \mathit{If}\ \ \ :: \mathsf{Stat\ Logic_s} \rightarrow \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ a} \\
\quad \mathit{Write} :: \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ ()} \\
\quad \mathit{Seq}\ \ :: \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ b} \rightarrow \mathsf{Stat\ b}
\end{array}$$

The new aspect here is the ability to give each constructor a different result type of the form Stat x. 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:

$$
\begin{array}{lll}
eq_{Stat} :: \mathsf{Stat\ a} \rightarrow \mathsf{Stat\ b} \rightarrow \mathsf{Bool} & & \\
eq_{Stat}\ (Val\ x_1) & (Val\ x_2) & = eq_{Expr}\ (==)\ x_1\ x_2 \\
eq_{Stat}\ (Term\ x_1) & (Term\ x_2) & = eq_{Logic}\ x_1\ x_2 \\
eq_{Stat}\ (If\ x_1\ x_2\ x_3) & (If\ x_1'\ x_2'\ x_3') & = eq_{Stat}\ x_1\ x_1' \wedge eq_{Stat}\ x_2\ x_2' \wedge eq_{Stat}\ x_3\ x_3' \\
eq_{Stat}\ (Write\ x_1) & (Write\ x_2) & = eq_{Stat}\ x_1\ x_2 \\
eq_{Stat}\ (Seq\ x_1\ x_2) & (Seq\ x_1'\ x_2') & = eq_{Stat}\ x_1\ x_1' \wedge eq_{Stat}\ x_2\ x_2' \\
eq_{Stat}\ \_ & \_ & = False
\end{array}
$$

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

Recently, an extensive comparison of generic programming libraries has been performed [Rodriguez et al., 2008b, Rodriguez, 2009]. 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 [Hinze and Löh, 2009]: 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.

Each of the libraries encodes the basic ingredients in a particular way. However, an encoding of a generic view on datatypes is largely orthogonal to an encoding of overloading, and we can achieve variants of the libraries described in the following sections by combining the basic ingredients differently [Hinze and Löh, 2009].

5 Lightweight Implementation of Generics and Dynamics

In this section, we discuss our first library for datatype-generic programming in Haskell. The library, Lightweight Implementation of Generics and Dynamics [Cheney and Hinze, 2002] or LIGD, serves as a good introduction to many of the concepts necessary for generic programming in a library. For example, it uses a simple encoding of the structural elements for the sum-of-products view that we saw in Section 2.7. Also, LIGD can represent many of the datatypes described in Section 3 with the exceptions being existentially quantified types and generalized algebraic datatypes (GADTs). Cheney and Hinze [2002] demonstrate a method for storing dynamically typed values (such as the one in Section 3.6); however, here we focus only on the generic representation. Lastly, we have updated the representation previously presented to use a GADT for type safety. As a side effect, it provides a good example of the material in Section 3.7.

To initiate our discussion of LIGD in Section 5.1, we first introduce an example function, in this case equality, to give a taste of how the library works. Then, Section 5.2 delves into the most basic representation used for LIGD. Next in Section 5.3, we show the important component necessary to support translating between the representation and Haskell datatypes. In Section 5.4, we describe how to implement a function differently for a certain type using overloading. Finally, Section 5.5 describes a number of useful generic functions (and how the library supports them), and Section 5.6 describes a particular case study using an exercise assistant.

5.1 An Example Function

The equality function in LIGD takes three arguments: the two values for comparison and a representation of the type of these values.

$$eq :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$$

The function *eq* is defined by pattern matching on the type representation type Rep, which contains constructors for type representations, such as *RInt*, *RChar*, etc. It is defined in the following subsection.

$$
\begin{array}{llll}
eq\ (RInt\ \ )\ & i & j & = eq_{Int}\ i\ j \\
eq\ (RChar) & c & d & = eq_{Char}\ c\ d \\
eq\ (RUnit) & Unit & Unit & = True \\
eq\ (RSum\ r_a\ r_b)\ & (L\ a_1) & (L\ a_2) & = eq\ r_a\ a_1\ a_2 \\
eq\ (RSum\ r_a\ r_b)\ & (R\ b_1) & (R\ b_2) & = eq\ r_b\ b_1\ b_2 \\
eq\ (RSum\ r_a\ r_b)\ & \_ & \_ & = False \\
eq\ (RProd\ r_a\ r_b)\ & (a_1 :\times: b_1) & (a_2 :\times: b_2) & = eq\ r_a\ a_1\ a_2 \wedge eq\ r_b\ b_1\ b_2
\end{array}
$$

Notice the similarities to the Generic Haskell function defined in Section 2.7. We have unit, sum, and product types, and the function is indexed by a representation of them, in this case the GADT Rep. By pattern matching on the constructors of Rep, the type checker is informed of the types of the remaining arguments, thus allowing us to pattern match on the structural elements.

Let us look at *eq* on a case-by-case basis. First, we have the primitive types Int and Char. These are not represented generically; rather, their values are stored, and we depend on the primitive functions eq_{Int} and eq_{Char}. Next, we have the collection of generic structural elements: unit, sum, and product. Two *Unit* values are always equal (ignoring undefined values). A sum presents two alternatives, *L* and *R*. If the structure is the same, then we recursively check equality of the contents; otherwise, the alternatives cannot be equal. In the last case, a product is only equal to another product if their components are both equal.

5.2 Run-Time Type Representation

The *eq* function is a type-indexed function with a run-time type representation as its first argument — it need not appear in that position, but that is standard practice. The representation utilizes a few key types for structure.

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

infixr 5 :+:
infixr 6 :×:
```

These three types represent the values of the unit, sum, and product, and each is isomorphic to a standard Haskell datatype: Unit to $()$, $(:+:)$ to Either, and $(:\times:)$ to $(,)$. We use new datatypes so as to easily distinguish the world of type representations and world of types we want to represent.

The GADT Rep uses the above datatypes to represent the structure of types.

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

The constructors in Rep define the universe of LIGD: the structural elements together with basic types. Of course, there are other basic types such as Float and Double, but their use is similar, and we ignore them for brevity.

Cheney and Hinze [2002] developed the original LIGD before GADTs had been introduced into GHC. They instead used an existentially quantified datatype. Using a GADT has the advantage that case analysis on types can be implemented by pattern matching, a familiar construct to functional programmers.

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 many 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 order to add arbitrary datatypes to the LIGD universe, we extend Rep with the *RType* constructor.

```
data Rep t where
  ...
  RType :: EP d r → Rep r → Rep d
```

The type r provides the structure representation for some datatype d. This indicates that r is isomorphic to d, and the isomorphism is witnessed by an embedding-projection pair.

```
data EP d r = EP{from :: (d → r), to :: (r → d)}
```

The type EP is a pair of functions for converting d values to r values and back. An EP value should preserve the properties (as described in Section 2.5) that $from \,.\, to \equiv id$ and $to \,.\, from \equiv id$.

As mentioned in Section 2.7, we can represent constructors by nested sums and fields by nested products. To give an example, the isomorphic representation type for List a is:

```
type RList a = Unit :+: a :×: List a
```

The functions for the embedding-projection are:

```
fromList :: List a → RList a
fromList Nil          = L Unit
```

```
from_List (Cons a as)     = R (a :×: as)

to_List :: RList a → List a
to_List (L Unit)          = Nil
to_List (R (a :×: as))    = Cons a as
```

The components of the pair are not embedded in the universe. The reason for this is that LIGD does not model recursion explicitly. This is sometimes called a *shallow representation*. In LIGD, a structure representation type is expressed in terms of the basic type representation types Int, Char, Unit, (:+:), and (:×:), and it may refer back to the type that is represented, argument types, and other types that have been represented. As a consequence, it is easy to represent mutually recursive datatypes as introduced in Section 3.3. Some generic programming libraries, such as PolyLib [Norell and Jansson, 2004a], use a *deep representation* of datatypes, in which the arguments of the structure representation types are embedded in the universe as well. This makes it easier to define some generic functions, but much harder to embed families of datatypes and mutually recursive datatypes. However, the very recent generic programming library multirec [Rodriguez et al., 2009] shows how to overcome this limitation.

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

```
r_List :: Rep a → Rep (List a)
r_List r_a = RType (EP from_List to_List)
                   (RSum RUnit (RProd r_a (r_List r_a)))
```

Given the definition of equality in Section 5.1, we can now extend it to support all representable types.

```
eq :: Rep a → a → a       → Bool
...
eq (RType ep r_a) t_1 t_2 = eq r_a (from ep t_1) (from ep t_2)
```

This case takes arguments t_1 and t_2 of some type a, transforms them to their structure representation using the embedding-projection pair *ep*, and applies equality to the new values with the representation r_a. Adding this line to the definition of *eq* turns it from a type-indexed function into a generic function.

Note that there are two ways to extend the LIGD universe to a type T. A non-generic extension involves adding a type-specific, ad-hoc constructor to Rep while a generic-extension requires a structure representation for T but no additional function cases. For example, support for Int is non-generic, and support for List is generic. The ability for generic extension is the feature that distinguishes generic functions from type-indexed functions.

Exercise 1. Give the representation of the datatypes Tree and Forest (defined in Section 3.3) for LIGD. ■

5.4 Support for Overloading

Now that we have seen a very basic generic function, we will explore a few other concepts of generic programming in LIGD. A "show" function — serving the same purpose

as the standard *show* function after **deriving** *Show* — illustrates how a library deals with constructor names and how it deals with ad-hoc cases for particular datatypes. First, we look at constructor names.

The type representation as developed so far does not contain any information about constructors, and hence we cannot define a useful generic show function using this representation. To solve this, we add an extra constructor to the structure representation type.

```
data Rep t where
  ...
  RCon :: String → Rep a → Rep a
```

To use this extra constructor, we modify the representation of the datatype List to include the names of the constructors:

```
r_List :: Rep a → Rep (List a)
r_List r_a = RType (EP from_List to_List)
                   (RSum (RCon "Nil" RUnit)
                         (RCon "Cons" (RProd r_a (r_List r_a))))
```

Here is a simple definition for a generic show function:

```
show :: Rep t → t → String
show RInt            t       = show t
show RChar           t       = show t
show RUnit           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 :×: 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)    t       = "(" ++ s ++ "  " ++ show r_a t ++ ")"
```

As an example of how *show* works, given an input of (*Cons* 1 (*Cons* 2 *Nil*)), it outputs `"(Cons 1 (Cons 2 Nil))"`. This definition works well generically, but the output for lists seems rather verbose. Suppose we want the list to appear in the comma-delimited fashion of the built-in Haskell lists, e.g. `"[1,2]"`. We can do that with an ad-hoc case for List.

For each type for which we want a generic function to behave in a non-generic way, we extend Rep with a new constructor. For lists, we add *RList*:

```
data Rep t where
  ...
  RList :: Rep a → Rep (List a)
```

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

```
show (RList r_a) as = show_List (show r_a) True as
```

This case uses the following useful higher-order function:

```
show_List :: (a → String) → Bool → List a → String
show_List show_a = go
  where go False Nil          = "]"
        go True  Nil          = "[]"
        go False (Cons a as) = ',' : rest a as
        go True  (Cons a as) = '[' : rest a as
        rest a as = show_a a ++ go False as
```

Now, *show* (*RList RInt*) (*Cons* 1 (*Cons* 2 *Nil*)) will print a nicely reduced list format. Note that the resulting generic function does not implement all details of **deriving** *Show*, but it does provide the core functionality.

We adapted the type representation Rep to obtain type-specific behavior in the *gshow* function. In general, it is undesirable to change a library in order to obtain special behavior for a single generic function on a particular datatype. Unfortunately, this is unavoidable in LIGD: for any generic function that needs special behavior on a particular datatype, we have to extend the type representation with that datatype. This means that users may decide to construct their own variant of the LIGD library, thus 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 disappears.

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. We can generate an "empty" value for any datatype representable by LIGD. For example, the empty value of Int is 0, and the empty value of List is *Nil*. The *empty* function encodes these choices.

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

Note that some of these choices are somewhat arbitrary. We might have used *minBound* for Int or *R* for sums.

An interesting aspect of this function is that it has a generic value as an output instead of an input. Up to now, we have only seen generic *consumer* functions or functions that accept generic arguments. A *producer* function constructs a generic value.

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. We previously introduced container datatypes in Sections 2.3 and 3.4. A useful function on a container datatype is a "flatten" function, which takes a value of the datatype and returns a list containing all values that it contains. For example, on the datatype Tree a, a flatten function would have type Tree a → [a]. We explain how to define this generic function in LIGD.

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

```
flatten :: Rep f → f a → [a]   -- WRONG!
```

where f abstracts over types of kind ⋆ → ⋆. Since Rep expects arguments of kind ⋆, this gives a kind error. Replacing Rep f by Rep (f a) would solve the kinding problem, but introduce 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 the type representation Rep (f a). We solve this problem by creating a new structure representation type:

```
data Rep1 g a where
  RInt1   ::                                  Rep1 g Int
  RChar1  ::                                  Rep1 g Char
  RUnit1  ::                                  Rep1 g Unit
  RSum1   :: Rep1 g a → Rep1 g b →            Rep1 g (a :+: b)
  RProd1  :: Rep1 g a → Rep1 g b →            Rep1 g (a :×: b)
  RType1  :: EP d r → Rep1 g r →              Rep1 g d
  RCon1   :: String → Rep1 g a →              Rep1 g a
  RVar1   :: g a →                            Rep1 g a
```

This datatype is very similar to Rep, but there are two important differences. The first is that Rep1 is now parametrised over two types: a generic function signature g of kind ⋆ → ⋆ and a generic type a of kind ⋆. The second change is the addition of the *RVar1* constructor. The combination of the signature, represented by a **newtype**, and the constructor *RVar1* will be used to define the functionality at occurrences of the type argument in constructors.

Our initial challenge for defining *flatten* is to choose a signature (for g above). In general, it should be the most general signature possible, and in our case, we note that our function takes one generic value and produces a list of non-generic elements. Thus, we know the following: it is a function with one argument, that argument is generic, and

the return value is a polymorphic list. From this information, we decide on the following **newtype** as our signature:

```
newtype Flatten b a = Flatten{selFlatten :: a → [b]}
```

It is important to notice that the order of the type parameters is significant. Flatten will be used as a type of kind $\star \to \star$, so the last parameter (a) serves as the generic argument type while the first parameter (b) is simply polymorphic.

Once we have our signature, we can define a type-indexed function (with a type synonym to improve the readability and reduce the visual complexity of types).

```
type RFlatten b a = Rep1 (Flatten b) a

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

The function *appFlatten* is not the final result, but it encompasses all of the structural induction on the representation. The primitive types and unit are not important to the structure of the container, so we return empty lists for them. In the sum case, we simply recurse to the appropriate alternative. For products, we append the second list of elements to the first list. In the *RType* case, we convert a Haskell datatype to its representation before recursing. Perhaps the most interesting case is *RVar1*.

The *RVar1* constructor tells us where to apply the function wrapped by our **newtype** signature. Thus, we select the function with the record destructor *selFlatten* and apply it to the value. Since we have not yet defined that signature function, our definition is not yet complete. We can define the signature function and the final result in one go:

```
flatten :: (RFlatten a a → RFlatten a (f a)) → f a → [a]
flatten rep = appFlatten (rep (RVar1 (Flatten (:[]))))
```

We have added a convenience to the type signature that is perhaps not obvious: it is specialized to take an argument of f a rather than the more general, single-variable type that would be inferred. This change allows us to look at the type and better predict the meaning of the function.

There are a few points worth highlighting in the definition of *flatten*. First, the type signature indicates that its first argument is a representation for a datatype of kind $\star \to \star$. This is evident from the functional type of the argument. Second, we see a value-level parallel to a type-level operation: the *rep* argument, representative of a type constructor, is applied to the *RVar1* value, itself standing in for the argument of a type

constructor. Lastly, our signature function is established by *Flatten* $(:[])$, where $(:[])$ injects an element into a singleton list. Notice the connection to *appFlatten* in which we use *selFlatten* to apply the signature function in the *RVar1* case.

Now, to see how *flatten* can be used, we create a representation for the List datatype using Rep1. When comparing with the previous representation for Rep, the constructor names and the type require trivial changes.

$$
\begin{aligned}
&r_{List,1} :: \mathsf{Rep1\ g\ a} \to \mathsf{Rep1\ g\ (List\ a)} \\
&r_{List,1}\ r_a = \mathit{RType1}\ (\mathit{EP}\ \mathit{from}_{List}\ \mathit{to}_{List}) \\
&\qquad\qquad (\mathit{RSum1}\ (\mathit{RCon1}\ \texttt{"Nil"}\ \mathit{RUnit1}) \\
&\qquad\qquad\qquad (\mathit{RCon1}\ \texttt{"Cons"}\ (\mathit{RProd1}\ r_a\ (r_{List,1}\ r_a))))
\end{aligned}
$$

We use this representation to produce a function specialized for lists:

$$
\begin{aligned}
&\mathit{flattenList} :: \mathsf{List\ a} \to [\mathsf{a}] \\
&\mathit{flattenList} = \mathit{flatten}\ r_{List,1}
\end{aligned}
$$

Of course, this transformation is isomorphic and not extremely useful, but we can apply the same approach to Tree and Forest for a more productive specialization.

Exercise 3. Many generic functions follow the same 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 logical "or" of all boolean values in a container. We implement this pattern with *crush*.

The function *crush* 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 *crush* follow.

$$
\begin{aligned}
&\mathbf{newtype}\ \mathsf{Crush\ b\ a} = \mathit{Crush}\{\mathit{gCrush} :: \mathsf{a} \to \mathsf{b}\} \\
&\mathit{crush} :: \mathsf{Rep1\ (Crush\ b)\ a} \to (\mathsf{b} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{a} \to \mathsf{b}
\end{aligned}
$$

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

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

Generalised Map. A well-known function is $\mathit{map} :: (\mathsf{a} \to \mathsf{b}) \to [\mathsf{a}] \to [\mathsf{b}]$ function. It takes a higher-order function and a list as arguments, and applies the function to every element in the list. We can also defined a generic *map* function that applied a function to every element of some container datatype. The *map* function can be viewed as the implementation of **deriving** *Functor*.

As with the generic *flatten*, the generic *map* function needs to know where the occurrences of the type argument of the datatype appear in a constructor. This means that we again need to abstract over type constructors. If we use Rep1 for our representation, the argument function will only return a value of a type that dependent on a or a constant type. Recall that the constructor *RVar1* has type $\mathsf{g\ a} \to \mathsf{Rep1\ g\ a}$, and thus the signature function g can only specify behavior for a single type variable. A true, generic *map*

should be able to change each element type from a to a possibly completely different type b; so, we need a signature function with two type variables.

Our generic *map* will use this new representation datatype.

```
data Rep2 g a b where
  RInt2   ::                                                  Rep2 g Int Int
  RChar2  ::                                                  Rep2 g Char Char
  RUnit2  ::                                                  Rep2 g Unit Unit
  RSum2   :: Rep2 g a b → Rep2 g c d →                        Rep2 g (a :+: c) (b :+: d)
  RProd2  :: Rep2 g a b → Rep2 g c d →                        Rep2 g (a :×: c) (b :×: d)
  RType2  :: EP a c → EP b d → Rep2 g c d →                   Rep2 g a b
  RCon2   :: String → Rep2 g a b →                            Rep2 g a b
  RVar2   :: g a b →                                          Rep2 g a b
```

The significant difference with the representation type Rep1 is the addition of the type variable b in Rep2 g a b and in the signature function g a b argument of *RVar2*. As we would expect, a signature function now has the kind $\star \to \star \to \star$. One other minor but necessary difference from Rep1 (and Rep) is the second EP argument to *RType*. Since we have two generic type parameters, we need an isomorphism for each.

We begin defining the generic function *map* with the signature function type as we did with *flatten*. Analyzing the problem we want to solve, we know that *map* requires a generic input value and a generic output value. There are no polymorphic or known types involved. So, our signature function is as follows:

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

Unlike the *flatten* example, the position of the parameters is not as important.

The type-indexed function appears as so:

```
type RMap a b = Rep2 Map a b

appMap :: RMap a b → a → b
appMap RInt2          i          = i
appMap RChar2         c          = c
appMap RUnit2         Unit       = Unit
appMap (RSum2 r_a r_b)  (L a)    = L (appMap r_a a)
appMap (RSum2 r_a r_b)  (R b)    = R (appMap r_b b)
appMap (RProd2 r_a r_b) (a :×: b) = appMap r_a a :×: appMap r_b b
appMap (RType2 ep1 ep2 r_a) x    = (to ep2 . appMap r_a . from ep1) x
appMap (RCon2 _ r_a)  x          = appMap r_a x
appMap (RVar2 f)      x          = selMap f x
```

Its definition is no real surprise. Since we only apply a change to elements of the container, we only use the signature function *selMap f* in the *RVar2* case. In every other case, we preserve the same structure on the right as on the left. It is also interesting to note the *RType2* case in which we translate *from* a datatype to its structure representation, apply the recursion, and translate the result back *to* the datatype.

The final part of the definition is quite similar to that of *flatten*.

$$map :: (\mathsf{RMap\ a\ b} \to \mathsf{RMap\ (f\ a)\ (f\ b)}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$$
$$map\ rep\ f = appMap\ (rep\ (RVar2\ (Map\ f)))$$

A major point of difference here is that the signature function f is an argument.

The representation of lists using this new representation type changes in insignificant ways: a second embedding-projection pair and naming updates.

$$\begin{aligned} r_{List,2}\ r_a = RType2\ &(EP\ from_{List}\ to_{List}) \\ &(EP\ from_{List}\ to_{List}) \\ &(RSum2\ (RCon2\ \texttt{"Nil"}\ RUnit2) \\ &\qquad (RCon2\ \texttt{"Cons"}\ (RProd2\ r_a\ (r_{List,2}\ r_a)))) \end{aligned}$$

Using $r_{List,2}$, we can define *map* on lists as follows:

$$mapList :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{List\ a} \to \mathsf{List\ b}$$
$$mapList = map\ r_{List,2}$$

Since each of the last two generic functions introduced required a new structure representation type, one might wonder if this happens for many generic functions. As far as we have found, the useful extensions stop with three generic type parameters. We could use the datatype Rep3 for all generic functions, but that would introduce many type variables that are never used. We prefer to use the representation type most suitable to the generic function at hand.

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

5.6 Case Study: Exercise Assistants

In this section, we describe using the LIGD library to define a generic function for a particular case study, an exercise assistant. An exercise assistant supports interactively solving exercises in a certain domain. For example, at the Open University NL and Utrecht University, we are developing exercise assistants for several domains: systems of linear equations [Passier and Jeuring, 2006], disjunctive normal form (DNF) of a logical expression [Lodder et al., 2006], and several kinds of exercises with 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 operations such as equality, rewriting, exercise generation, term traversal, selection, and serialization. Each program can be viewed as an instance of a generic exercise assistant. For each generic programming library we discuss in these lecture notes, we also present a case study implementing functionality for an exercise assistant. In this subsection, we show how to implement a generic function for determining the difference between two terms.

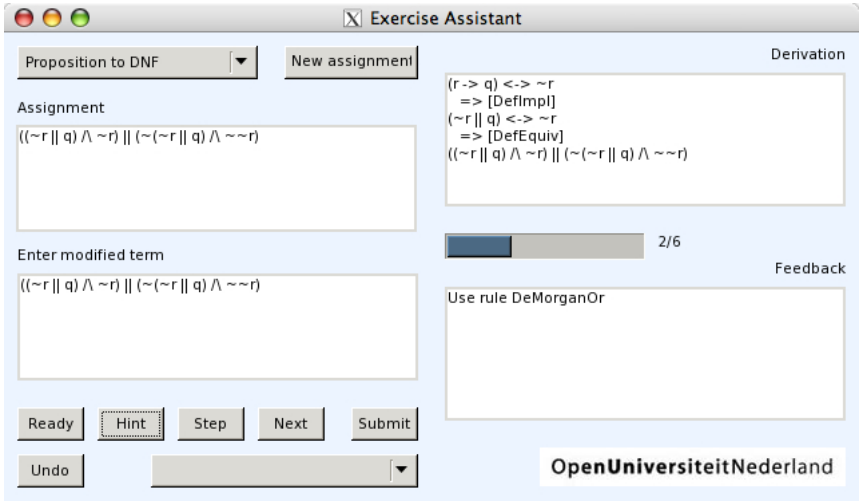

Fig. 1. The Exercise Assistant

We have used equality to introduce many concepts in these notes; however, we often want to know whether or not two values differ, and by *how much* or *where*. 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 match, we want to find a correctly rewritten expression that is closest in some sense to the expression submitted by the student.

The function *similar* 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 traverse deeper into the children.

$$
\begin{array}{llll}
\mathit{similar} :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Int} & & & \\
\mathit{similar}\ \mathit{RInt} & i & j & = \mathbf{if}\ i == j\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\mathit{similar}\ \mathit{RChar} & c & d & = \mathbf{if}\ c == d\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
\mathit{similar}\ \mathit{RUnit} & \_ & \_ & = 1 \\
\mathit{similar}\ (\mathit{RSum}\ r_a\ r_b) & (L\ a) & (L\ b) & = \mathit{similar}\ r_a\ a\ b \\
\mathit{similar}\ (\mathit{RSum}\ r_a\ r_b) & (R\ a) & (R\ b) & = \mathit{similar}\ r_b\ a\ b \\
\mathit{similar}\ (\mathit{RSum}\ r_a\ rA\_) & \_ & \_ & = 0 \\
\mathit{similar}\ (\mathit{RProd}\ r_a\ r_b) & (a_1 :\times: b_1) & (a_2 :\times: b_2) & = \mathit{similar}\ r_a\ a_1\ a_2 + \mathit{similar}\ r_b\ b_1\ b_2 \\
\mathit{similar}\ (\mathit{RType}\ ep\ r_a) & a & b & = \mathit{similar}\ r_a\ (\mathit{from}\ ep\ a)\ (\mathit{from}\ ep\ b) \\
\mathit{similar}\ (\mathit{RCon}\ \ s\ r_a) & a & b & = 1 + \mathit{similar}\ r_a\ a\ b
\end{array}
$$

Given a definition of a generic function *size* that returns the size of a value by counting all basic values and constructors, we can define the function *diff* by:

$$
\begin{array}{l}
\mathit{diff} :: \mathsf{Rep\ a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Int} \\
\mathit{diff}\ \mathit{rep}\ x\ y = \mathit{size}\ \mathit{rep}\ x - \mathit{similar}\ \mathit{rep}\ x\ y
\end{array}
$$

The difference here is reported as the size of the first value minus the similarity of the two. The function *diff* provides a rough estimate. A generic minimum edit distance function [Lempsink et al., 2009] would provide a higher-precision difference.

In this section, we discussed an implementation of datatype-generic programming in the Lightweight Implementation of Generics and Dynamics library. In the next section, we discuss a library that is similar in representation to LIGD but uses type classes instead of GADTs.

6 Extensible and Modular Generics for the Masses

The library "Generics for the Masses" was first introduced by Hinze [2004], and a variant, "Extensible and Modular Generics for the Masses," was later presented by Oliveira et al. [2006]. In this section, we describe latter, EMGM, with a slight twist to ease the extensibility requirements (details in Section 6.6).

Our approach follows much like that of Section 5. We again use equality to introduce generic functions (Section 6.1). We also explain the general mechanics (Section 6.2), the component necessary for extending the universe (Section 6.3), and the support for overloading (Section 6.4). Where EMGM differs from LIGD is the capability for generic functions to be extended with datatype-specific functionality while preserving the modularity of the function definition. We first describe the published approach to solving this problem (Section 6.5) and then introduce our solution to reducing the burden of extensibility (Section 6.6). Next, we define several different generic functions using EMGM (Section 6.7). As with LIGD, these require changes to the representation. Finally, we implement a value generator for the exercise assistant case study (Section 6.8).

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.

$$\textbf{newtype}\ \mathsf{Eq\ a} = Eq\{\, selEq :: \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}\,\}$$

The **newtype** Eq serves a similar purpose to the signature function of LIGD first mentioned when describing the function *flatten* in Section 5.5. Unlike LIGD, however, every generic function in EMGM requires its own **newtype**.

Next, we define the cases of our generic function.

$$\begin{array}{llll}
selEq_{int} & i & j & = i == j \\
selEq_{char} & c & d & = c == d \\
selEq_{\mathbb{1}} & Unit & Unit & = True \\
selEq_{+}\ r_a\ r_b\ (L\ a_1) & & (L\ a_2) & = selEq\ r_a\ a_1\ a_2 \\
selEq_{+}\ r_a\ r_b\ (R\ b_1) & & (R\ b_2) & = selEq\ r_b\ b_1\ b_2 \\
selEq_{+}\ _\ _\ _ & & _ & = False \\
selEq_{\times}\ r_a\ r_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & & & = selEq\ r_a\ a_1\ a_2 \wedge selEq\ r_b\ b_1\ b_2
\end{array}$$

We can read this in the same fashion as a type-indexed function in LIGD. Indeed, there is a high degree of similarity. However, instead of a single function that uses pattern matching on a type representation, we have many functions, each corresponding to a primitive or structural type. Another major difference with LIGD is that the type representation parameters (e.g. for *RSum*, *RProd*, etc.) are explicit and not embedded in the Rep datatype. Specifically, each function takes the appropriate number of representations according to the arity of the structural element. For example, $selEq_{\mathbb{1}}$ has no representation arguments, and $selEq_{+}$ and $selEq_{\times}$ each have two.

These functions are only part of the story, of course. Notice that $selEq_{+}$ and $selEq_{\times}$ 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 of a type class *Generic* for Eq:

```
instance Generic Eq where
  rint        = Eq selEq_int
  rchar       = Eq selEq_char
  runit       = Eq selEq_1
  rsum ra rb  = Eq (selEq_+ ra rb)
  rprod ra rb = Eq (selEq_× ra rb)
```

The type class has member functions corresponding to primitive and structure types. Each method defines the instance of the type-indexed function for the associated type. The above collection of functions are now used in 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. Another difference between the two libraries is that LIGD uses explicit recursion while EMGM's recursion is implicitly implemented by the instance in a fold-like manner.

We now have all of the necessary parts to use the type-indexed function *selEq*.[4]

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

On the other hand, we should not need to provide an explicit representation every time. Instead, we introduce a convenient wrapper that determines which type representation we need.

```
eq :: (Rep a) ⇒ a → a → Bool
eq = selEq rep
```

The type class *Rep* is an interface (Section 2.4) to all known type representations, and its method *rep* statically resolves to a value of the appropriate representation. This mechanism allows us to write a simpler call: `eq ('Q' :×: 42) ('Q' :×: 42)`. Note that we might have defined such a class for LIGD (as was done by Cheney and Hinze [2002]); however, that would have only been a convenience. In EMGM, it becomes a necessity for extensibility (Section 6.5).

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

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 6.7).

The type class *Generic* serves as the interface for a generic function.

```
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)

infixr 5 `rsum`
infixr 6 `rprod`
```

The class is parametrised by the type constructor g that serves as the type-indexed function's signature function.

Each method of the class represents a case of the type-indexed 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 (see Section 2.4). Thus, in our previous example, each call to *selEq* may have a different type.

The type-indexed function as we have defined it to this point is a destructor for the type g. As such, it requires an value 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
```

This allows us to replace any value of the type g a with *rep*. This simple but powerful concept uses the type system to dispatch the necessary representation. Representation instances are 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* given the argument $'\mathsf{Q}' :\times: (42 :: \mathsf{Int})$. 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 → g a → 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 (Section 5.3).

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) ⇒ g a → 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) ⇒ 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 $\mathsf{Logic_L}$ and $\mathsf{Logic_F}$ from Section 3.4. 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 in Section 5.4, we illustrate this support using a generic *show* function and lists and strings.

For accessing constructor names in the definition of a generic function, we add another method to our generic function interface.

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

We use *rcon* to label other structure components with a constructor name[5]. 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 function:

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

selShow_int          i          = show i
selShow_char         c          = show c
selShow_𝟙            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
selShow_type ep ra   a          = selShow ra (from ep a)
selShow_con  s  ra   a          = "(" ++ s ++ " " ++ selShow ra a ++ ")"

instance Generic Show where
  rint          = Show selShow_int
```

[5] The released EMGM library uses ConDescr instead of String. ConDescr contains a more comprehensive description of a constructor (fixity, arity, etc.). For simplicity's sake, we only use the constructor name in our presentation.

$$
\begin{array}{lll}
\mathit{rchar} & & = \mathit{Show}\ \mathit{selShow}_{\mathit{char}} \\
\mathit{runit} & & = \mathit{Show}\ \mathit{selShow}_{\mathbb{1}} \\
\mathit{rsum} & r_a\ r_b & = \mathit{Show}\ (\mathit{selShow}_{+}\ r_a\ r_b) \\
\mathit{rprod} & r_a\ r_b & = \mathit{Show}\ (\mathit{selShow}_{\times}\ r_a\ r_b) \\
\mathit{rtype}\ \mathit{ep}\ r_a & & = \mathit{Show}\ (\mathit{selShow}_{\mathit{type}}\ \mathit{ep}\ r_a) \\
\mathit{rcon}\ \ s\ \ r_a & & = \mathit{Show}\ (\mathit{selShow}_{\mathit{con}}\ s\ r_a)
\end{array}
$$

$$
\begin{array}{l}
\mathit{show} :: (\mathit{Rep}\ \mathsf{a}) \Rightarrow \mathsf{a} \rightarrow \mathsf{String} \\
\mathit{show} = \mathit{selShow}\ \mathit{rep}
\end{array}
$$

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

$$\mathit{show}\ (\mathit{Cons}\ 5\ (\mathit{Cons}\ 3\ \mathit{Nil})) \rightsquigarrow \texttt{"(Cons 5 (Cons 3 (Nil)))"}$$

As mentioned in Section 5.4, we would prefer to see this list as it natively appears in Haskell: `"[5,3]"`. To this end, just as we added a *RList* constructor to the Rep GADT, it is possible to add a method *rlist* to *Generic*.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Generic}\ \mathsf{g}\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{rlist} :: \mathsf{g\ a} \rightarrow \mathsf{g}\ (\mathsf{List\ a})
\end{array}
$$

It is then straightforward to define a new case for the generic show function, reusing the $\mathit{show}_{\mathit{List}}$ function from Section 5.4.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Generic}\ \mathsf{Show}\ \textbf{where} \\
\quad \ldots \\
> \mathit{rlist}\ r_a = \mathit{Show}\ (\mathit{show}_{\mathit{List}}\ (\mathit{selShow}\ r_a)\ \mathit{True})
\end{array}
$$

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.

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Rep}\ \mathsf{a}) \Rightarrow \mathit{Rep}\ (\mathsf{List\ a})\ \textbf{where} \\
\quad \mathit{rep} = \mathit{rlist}\ \mathit{rep}
\end{array}
$$

Now, when applying the example application of *show* above, we receive the more concise output.

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. Unlike LIGD, there are solutions for preserving modularity while allowing extensibility.

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 = r_List
```

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_a = Show (show_List (selShow r_a) True)
```

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 $(Generic\ g, Rep\ a) \Rightarrow g\ a$. 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 function signature type g. We subsequently redefine *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
```

This approach of using a type-specific class (e.g. *GenericList*) for extensibility as described initially by Oliveira et al. [2006] and demonstrated here by us puts an extra burden on the user. In the next subsection, we explain the problem and how we rectify it.

6.6 Reducing the Burden of Extensibility

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 (empty) instance to 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 the Haskell language.

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) ⇒ 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) ⇒ Rep Show (List a) where
  rep = Show (show_List (selShow rep) True)
```

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

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

```
data Ordering = LT | EQ | GT
compare :: (Ord a) ⇒ a → a → 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 *compare* that behaves like **deriving** *Ord*. 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. As we did with LIGD in Section 5.5, we write the generic producer function *empty* in EMGM as follows:

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

instance Generic Empty where
  rint          = Empty 0
```

$$
\begin{array}{lll}
\mathit{rchar} & = & \mathit{Empty}\ \texttt{'\textbackslash NUL'} \\
\mathit{runit} & = & \mathit{Empty}\ \mathit{Unit} \\
\mathit{rsum}\quad r_a\ r_b & = & \mathit{Empty}\ (L\ (\mathit{selEmpty}\ r_a)) \\
\mathit{rprod}\quad r_a\ r_b & = & \mathit{Empty}\ (\mathit{selEmpty}\ r_a :\times: \mathit{selEmpty}\ r_b) \\
\mathit{rtype}\ \mathit{ep}\ r_a & = & \mathit{Empty}\ (\mathit{to}\ \mathit{ep}\ (\mathit{selEmpty}\ r_a)) \\
\mathit{rcon}\ \ s\ \ r_a & = & \mathit{Empty}\ (\mathit{selEmpty}\ r_a)
\end{array}
$$

$$
\begin{array}{l}
\mathit{empty} :: (\mathit{Rep}\ \mathsf{Empty\ a}) \Rightarrow \mathsf{a} \\
\mathit{empty} = \mathit{selEmpty}\ \mathit{rep}
\end{array}
$$

There are a two noteworthy differences from previous examples. First, since it is a producer function, *empty* outputs a generic value. Unlike *empty* in LIGD, however, the EMGM version takes no arguments at all. In order to use it, we need to specify a concrete type. In the case where this is not inferred, we can give a type annotation.

$$\mathit{empty} :: \mathsf{Int} :+: \mathsf{Char} \rightsquigarrow L\ 0$$

The second difference lies in the *rtype* definition, where we use *to ep* instead of *from ep*. This is also characteristic of producer functions.

Crush and Flatten. Crush is a fold-like operation over a container datatype [Meertens, 1996]. It is a very flexible function, and many other useful functions can be implemented using crush. As mentioned in Exercise 3, it can be used to implement *flatten*, which we will also demonstrate.

Our goal is a function with a signature similar to the following for datatypes of kind $\star \rightarrow \star$ (see discussion for *flatten* in Section 5.5).

$$\mathit{crushr} :: (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}) \rightarrow \mathsf{b} \rightarrow \mathsf{f\ a} \rightarrow \mathsf{b}$$

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 *foldr* function. 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.

$$\mathbf{newtype}\ \mathsf{Crushr\ b\ a} = \mathit{Crushr}\{\mathit{selCrushr} :: \mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}\}$$

This function extracts the container's element and combines it with a partial result to produce a final result. The implementation follows[6]:

$$
\begin{array}{lll}
\mathit{crushr}_{\mathit{int}} & \_ & e = e \\
\mathit{crushr}_{\mathit{char}} & \_ & e = e \\
\mathit{crushr}_{\mathbb{1}} & \_ & e = e
\end{array}
$$

[6] For brevity, we elide most of the *Generic* instance declaration. It is the same as we have seen before.

$$
\begin{array}{lllll}
crushr_{+} & r_a\ r_b\ (L\ a) & e = selCrushr\ r_a\ a\ e \\
crushr_{+} & r_a\ r_b\ (R\ b) & e = selCrushr\ r_b\ b\ e \\
crushr_{\times} & r_a\ r_b\ (a :\times: b) & e = selCrushr\ r_a\ a\ (selCrushr\ r_b\ b\ e) \\
crushr_{type}\ ep & r_a \quad a & e = selCrushr\ r_a\ (from\ ep\ a)\ e \\
crushr_{con}\ s & r_a \quad a & e = selCrushr\ r_a\ a\ e
\end{array}
$$

```
instance Generic (Crushr b) where
  rint = Crushr crushr_int
  ...
```

Note that *selCrushr* is only applied to the parametrised structural type cases: $crushr_{+}$, $crushr_{\times}$, $crushr_{type}$, 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 ⋆. We need a representation for a type of kind ⋆ → ⋆. 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 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 = r_List
```

Now we can define *crushr*; 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 definition 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 define the *flattenr* function as a specialization. Recall that flattening involves translating all elements of a structure into a list. The definition of *flattenr* requires only the combining operator, $(:)$, for inserting an element into a list and the zero value, $[\,]$, for starting a new list.

```
flattenr :: (FRep (Crushr [a]) f) ⇒ f a → [a]
flattenr = crushr (:) []
```

Exercise 7. How is the behavior of the EMGM function *crushr* different from that of the LIGD function *crush*? Why might the *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. 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:

```
map :: (a → b) → f a → f b
```

Using the same analysis performed to define the signature function for *map* in LIGD, we arrive at the same type.

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

This means we need to abstract over both type arguments in Map. We have not yet seen how that is done in EMGM, but the idea is similar to the change in LIGD's representation.

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)
  rprod2 :: g a1 a2 → g b1 b2 → g (a1 :×: b1) (a2 :×: b2)
  rtype2 :: EP a2 a1 → EP b2 b1 → g a1 b1 → g a2 b2
```

The major difference from *Generic* is that the signature function type g now has kind $\star \to \star \to \star \to \star$. In the case of the primitive types and Unit, this means simply repeating the type twice. In the case of $(:+:)$ and $(:\times:)$, we need to pass two types around instead of one. The method *rtype2*, like the constructor *RType2* now accepts two embedding-projection pairs.

The implementation of the generic function follows:

$$
\begin{array}{lll}
map_{int} & i & = i \\
map_{char} & c & = c \\
map_{\mathbb{1}} & Unit & = Unit \\
map_{+} & r_a\ r_b\ (L\ a) & = L\ (selMap\ r_a\ a) \\
map_{+} & r_a\ r_b\ (R\ b) & = R\ (selMap\ r_b\ b) \\
map_{\times} & r_a\ r_b\ (a :\times: b) & = selMap\ r_a\ a :\times: selMap\ r_b\ b \\
map_{type}\ ep_1\ ep_2\ r_a & x & = (to\ ep_2\ .\ selMap\ r_a\ .\ from\ ep_1)\ x
\end{array}
$$

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

The explanation for the implementation follows exactly as the one given for LIGD's *appMap* except for the *RVar2* case, which EMGM does not have.

We write the representation for list as:

```
r_List,2 :: (Generic2 g) => g a b -> g (List a) (List b)
r_List,2 ra = rtype2 (EP from_List to_List) (EP from_List to_List)
                     (runit2 'rsum2' ra 'rprod2' r_List,2 ra)
```

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 in the same way 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 signature function type g with two argument types. Note, however, that we still expect functionally kinded datatypes: f has kind $\star \to \star$.

Finally, we provide our definition of *map*.

$$map :: (FRep2\ \mathsf{Map\ f}) \Rightarrow (\mathsf{a} \to \mathsf{b}) \to \mathsf{f\ a} \to \mathsf{f\ b}$$
$$map = selMap\ .\ frep2\ .\ \mathsf{Map}$$

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:

$$zipWith :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to \mathsf{f\ a} \to \mathsf{f\ b} \to \mathsf{f\ c}$$

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

6.8 Case Study: Generating Values

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 would be duplicated for each domain. For the latter, we need to generically generate exercises. This leads us to defining a generic value generator.

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

$$gen :: \mathsf{Int} \to \mathsf{a}$$

gen takes a (possibly randomly generated) integer and returns a value somehow representative of that number. 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 becomes 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 template of a generic function:

$$\textbf{newtype}\ \mathsf{Enum\ a} = Enum\{selEnum :: [\mathsf{a}]\}$$
$$\textbf{instance}\ Generic\ \mathsf{Enum}\ \textbf{where}$$
$$\begin{aligned}
rint &= Enum\ enum_{int}\\
rchar &= Enum\ enum_{char}\\
runit &= Enum\ enum_{\mathbb{1}}\\
rsum\ \ r_a\ r_b &= Enum\ (enum_{+}\ \ r_a\ r_b)\\
rprod\ \ r_a\ r_b &= Enum\ (enum_{\times}\ \ r_a\ r_b)\\
rtype\ ep\ r_a &= Enum\ (enum_{type}\ ep\ r_a)\\
rcon\ \ s\ \ r_a &= Enum\ (enum_{con}\ s\ r_a)
\end{aligned}$$

Now, let us fill in each case of the function. 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:

$$enum_{int} = [minBound\,..\,maxBound]$$

However, that would lead to a (very long) list of negative numbers followed by another (very long) list of negative numbers. This is an awfully unbalanced sequence while we would prefer to start with the most "basic" value (equivalent to *empty*) and progressively get larger. As a result, we alternate positive and negative numbers.

$$enum_{int} = [0\,..\,maxBound] \,|||\, [-1,-2\,..\,minBound]$$

By reversing the negative enumeration, we now begin with 0 and grow larger (in the absolute sense). The interleave operator $(|||)$ is defined as follows:

$$\begin{array}{l}(|||) :: [\mathsf{a}] \to [\mathsf{a}] \to [\mathsf{a}] \\ [\,] \quad\;\; ||| \; ys = ys \\ (x : xs) \,|||\, ys = x : ys \,|||\, xs\end{array}$$

This function 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 $(|||)$ also works if the lists are infinite.

For Char and Unit, the implementations are straightforward.

$$\begin{array}{ll}enum_{char} & = [minBound\,..\,maxBound] \\ enum_{\mathbb{1}} & = [Unit]\end{array}$$

For $enum_{char}$, we enumerate from the first character `'\NUL'` to the last, and for $enum_{\mathbb{1}}$, we return a singleton list of the only Unit value.

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.

$$enum_{+}\; r_a\; r_b = [L\; x \mid x \leftarrow selEnum\; r_a] \,|||\, [R\; y \mid y \leftarrow selEnum\; r_b]$$

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.

$$enum_{\times}\; r_a\; r_b = selEnum\; r_a \mathbin{\times\!\!\!\!\!\!\times} selEnum\; r_b$$

The definition of $\times\!\!\!\!\!\!\times$ is left as Exercise 11 for the reader.

The remaining 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.

$$enum :: (Rep\ \mathsf{Enum\ a}) \Rightarrow [\mathsf{a}]$$
$$enum = selEnum\ rep$$

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

$$gen :: (Rep\ \mathsf{Enum\ a}) \Rightarrow \mathsf{Int} \rightarrow \mathsf{a}$$
$$gen = (!!)\ enum$$

The performance of this function is not be optimal; however, we could fuse the indexing operator $(!!)$ into the definition of *enum* for a more efficient (and more complicated) function.

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

$$diag :: [[\mathsf{a}]] \rightarrow [\mathsf{a}]$$

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.

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

■

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

We have provided an introduction to the Extensible and Modular Generics for the Masses library in this section. It relies on similar concepts to LIGD, yet it allows for better extensibility and modularity through the use of type classes. The next section introduces a well-known library using a representation that is completely different from both LIGD and EMGM.

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 original concept behind SYB is that in contrast to the two approaches discussed previously, the structure of datatypes is not directly exposed to the programmer. Generic functions are built with "primitive" generic combinators, and the combinators in turn can be generated (in GHC) using Haskell's **deriving** mechanism for type classes. We also mention a variation of SYB in which a structure representation is given and used to define functions.

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:

$$inc :: \mathsf{Expr\ Int} \rightarrow \mathsf{Expr\ Int}$$
$$inc\ (Lit\ x) = Lit\ (x + 1)$$

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:

$$increment :: Data\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{a}$$
$$increment = everywhere\ (mkT\ inc)$$

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 instances should not be defined 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 capabilities[7].

```
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)
             → 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 dualizing unfold. The types of these combinators may be a bit intimidating, and they are 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))
```

[7] The *Data* class has 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.

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.

$$\begin{aligned}
&gsize :: Data\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{Int}\\
&gsize = unBox\ .\ gfoldl\ k\ (\lambda_ \rightarrow IntBox\ 1)\ \mathbf{where}\\
&\quad k\ (IntBox\ h)\ t = IntBox\ (h + gsize\ t)\\
&\mathbf{newtype}\ \mathsf{IntBox\ x} = IntBox\{unBox :: \mathsf{Int}\}
\end{aligned}$$

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. The type parameter x of IntBox is a phantom type: it is not used as a value, but is necessary for type correctness.

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 consumes 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. It is technically not an unfold, but instead a fold on a different view [Hinze and Löh, 2006].

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 datatype.[8]

$$\begin{aligned}
&toConstr\ \ Nil\ \ \ \ \ \ \ \ = con_1\\
&toConstr\ \ (Cons\ _\ _) = con_2\\
&dataTypeOf\ _\ \ \ \ \ \ \ \ \ = ty\\
&dataCast1\ \ f\ \ \ \ \ \ \ \ \ \ = gcast1\ f\\
&con_1 = mkConstr\ ty\ \texttt{"Empty_List"}\ [\,]\ Prefix\\
&con_2 = mkConstr\ ty\ \texttt{"Cons_List"}\ \ [\,]\ Prefix\\
&ty\ \ \ = mkDataType\ \texttt{"ModuleNameHere"}\ [con_1, con_2]
\end{aligned}$$

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

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

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. Finally, since the List datatype is not of kind $\star$, we have to provide an implementation for the *dataCast1* method in terms of *gcast1*.[9]

SYB supports all datatypes for which we can give a *Data* and *Typeable* instance. This includes all datatypes of Section 3 except GADTs and existentially quantified types, for which we cannot define *gunfold*.

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.

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:

$$\textbf{type}\ \mathsf{GenericT} = \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{a}$$

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

$$\textbf{type}\ \mathsf{GenericM\ m} = \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{m\ a}$$

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

$$\textbf{type}\ \mathsf{GenericQ\ r} = \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{r}$$

A builder produces a value of a particular type.

$$\textbf{type}\ \mathsf{GenericB} = \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a}$$

[9] Datatypes of kind $\star \to \star$ require the definition of dataCast1, and datatypes of kind $\star \to \star \to \star$ require the definition of dataCast2. For datatypes of kind $\star$, the default definition for these methods (*const Nothing*) is appropriate.

A builder that has access to a monad is called a "reader".

$$\textbf{type}\ \mathsf{GenericR}\ \mathsf{m} = \forall \mathsf{a}\,.\,\mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{m}\ \mathsf{a}$$

Note however that the types of both GenericM and GenericR do 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:

$$\begin{aligned}&\mathit{everywhere} :: \mathsf{GenericT} \to \mathsf{GenericT}\\ &\mathit{everywhere}\ f = f\,.\,\mathit{gmapT}\ (\mathit{everywhere}\ f)\end{aligned}$$

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

$$\begin{aligned}&\mathit{gmapT} :: \mathit{Data}\ \mathsf{a} \Rightarrow (\forall \mathsf{b}\,.\,\mathit{Data}\ \mathsf{b} \Rightarrow \mathsf{b} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{a}\\ &\mathit{gmapT}\ f\ x = \mathit{unID}\ (\mathit{gfoldl}\ k\ \mathit{ID}\ x)\\ &\quad \textbf{where}\\ &\qquad k\ (\mathit{ID}\ c)\ y = \mathit{ID}\ (c\ (f\ y))\\ &\textbf{newtype}\ \mathsf{ID}\ \mathsf{x} = \mathit{ID}\{\mathit{unID} :: \mathsf{x}\}\end{aligned}$$

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

$$\mathit{everywhere}' :: \mathsf{GenericT} \to \mathsf{GenericT}$$

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.

$$\mathit{mkT} :: (\mathit{Typeable}\ \mathsf{a}, \mathit{Typeable}\ \mathsf{b}) \Rightarrow (\mathsf{b} \to \mathsf{b}) \to \mathsf{a} \to \mathsf{a}$$

For example, $\mathit{mkT}\ (\mathit{sin} :: \mathsf{Float} \to \mathsf{Float})$, 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.

$$\begin{aligned}&\mathit{total} :: \mathsf{GenericQ}\ \mathsf{Int}\\ &\mathit{total} = \mathit{everything}\ (+)\ (0\ \text{`}\mathit{mkQ}\text{'}\ \mathit{lit})\ \textbf{where}\\ &\quad \mathit{lit} :: \mathsf{Expr}\ \mathsf{Int} \to \mathsf{Int}\end{aligned}$$

$$\begin{aligned} &\mathit{lit}\ (\mathit{Lit}\ x) = x \\ &\mathit{lit}\ x \qquad\quad = 0 \end{aligned}$$

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 $\mathsf{a} \to \mathsf{b}$, together with a default value of type b, to a generic a query function of type $\mathsf{GenericQ\ b}$. 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*.

$$\begin{aligned} &\mathit{gsize} :: \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{Int} \\ &\mathit{gsize}\ t = 1 + \mathit{sum}\ (\mathit{gmapQ}\ \mathit{gsize}\ t) \\ &\mathit{glength} :: \mathsf{GenericQ\ Int} \\ &\mathit{glength} = \mathit{length} \,.\, \mathit{gmapQ}\ (\mathit{const}\ ()) \end{aligned}$$

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

$$\mathit{gmapQ} :: (\forall \mathsf{a} \,.\, \mathit{Data}\ \mathsf{a} \Rightarrow \mathsf{a} \to \mathsf{u}) \to \mathsf{a} \to [\mathsf{u}]$$

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:

$$\mathit{gdepth} :: \mathsf{GenericQ\ Int}$$

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:

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

■

Exercise 16. Define the function:

$$gwidth :: \mathsf{GenericQ\ Int}$$

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:

$$\begin{gathered} gwidth\ () \rightsquigarrow \\ gwidth\ (Just\ 1) \rightsquigarrow \\ gwidth\ ((1,2),(1,2)) \rightsquigarrow \\ gwidth\ (((1,2),2),(1,2)) \rightsquigarrow \end{gathered}$$

■

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.

$$zipWith :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to [\mathsf{a}] \to [\mathsf{b}] \to [\mathsf{c}]$$

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

$$gzipWithQ :: \mathsf{GenericQ\ (GenericQ\ c)} \to \mathsf{GenericQ\ (GenericQ\ [c])}$$

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 $\mathsf{a} \to \mathsf{b} \to \mathsf{c}$. 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 is apparent 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*.

```
gshow_s "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. It can be seen as a two-way case branch: if the input term (the last argument) is of type b, then the second function is applied. If not, then the first function is applied. 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 $\star \to \star$, 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 $\star \to \star \to \star$, 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 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:

$$gshow'\ (One\ \texttt{"a"}) \rightsquigarrow \texttt{"String: a"}$$

as we expected. However:

$$gshow'\ (One\ \texttt{"a"}, One\ \texttt{"b"}) \rightsquigarrow \texttt{"((,) (One \"a\") (One \"b\"))"}$$

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.

7.7 An Explicit View for SYB

Unlike in the two approaches we have seen before, the mechanism for run-time type representation in SYB does not involve an explicit generic view on data. Scrap Your Boilerplate Reloaded [Hinze et al., 2006] presents an alternative interpretation of SYB by replacing the combinator based approach 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
```

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 by[10] $Con\ (:) \diamond (Int\ :^:\ 1) \diamond (List\ Int\ :^:\ [2])$. We can define the usual SYB combinators

[10] Note the difference between the list constructor $(:)$ and the Typed constructor $(:^:)$.

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 *map*, in a natural way. Additionally, functions taking multiple arguments (such as generic equality) also become straightforward to define.

A disadvantage of having the explicit Spine view 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 instead. It is possible to make the variants extensible by using a similar approach as discussed in Section 7.6: abstraction over type classes. We refer the reader to [Hinze et al., 2006, Hinze and Löh, 2006] for further information.

7.8 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. This means that the student selects a range in a pretty-printed expression and chooses a rule to apply to the selection.

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 selection 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 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. For simplicity, we do not deal with associativity or infix constructors. We use a state monad

transformer with an embedded writer monad. The state monad keeps track of the current position using an Int, while 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 let Type be 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.

We can check that our function behaves as expected (for the $\mathsf{Logic_s}$ type of Section 3.4):

$$map\ fst\ .\ M.to_{List}\ .\ selections\ \$\ (Or\ (Not\ (Lit\ True))\ (Lit\ False)) \rightsquigarrow [(0,37),(4,22),(9,21),(14,20),(23,36),(28,35)]$$

Indeed we can confirm that

$(0,37)$ corresponds to $(Or\ (Not\ (Lit\ (True)))\ (Lit\ (False)))$
$(4,22)$ corresponds to $(Not\ (Lit\ (True)))$
$(9,21)$ corresponds to $(Lit\ (True))$
$(14,20)$ corresponds to $(True)$
$(23,36)$ corresponds to $(Lit\ (False))$
$(28,35)$ corresponds to $(False)$

As mentioned before, the *selections* function presented in this section has been simplified in many ways. Possible improvements include support for operator fixity and precedence (which change the parentheses), mapping a range to the actual value in the selection, dealing with associative operators and decoupling from a fixed pretty-printer (*gshow* in this case). Additionally, selections of constant types (such as Bool in the example above) are typically not relevant and should not be considered valid.

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 (introduced in Section 3.6). ■

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 9.1 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.

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 7.6).

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 it is across different Haskell compilers.

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, and LIGD is the most portable of the three libraries.

Generics for the Masses as originally introduced [Hinze, 2004] was entirely within Haskell 98; however, EMGM as described in these notes is not as portable. It relies on multiparameter type classes to support implicit type representations and type operators for convenience (both currently slated to become part of Haskell Prime). The features for supporting convenient extensibility (Sections 6.5 and 6.6) also rely on overlapping and undecidable instances, and we do not know if these will become part of Haskell Prime.

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[11].

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

[11] This has recently been shown by Reinke [2008] and Kiselyov [2008]. However, the definition is rather intricate, and as such we do not present it in these notes.

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 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.

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?

9.1 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, Hinze and Jeuring, 2003b]
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, 2000]
An Introduction to Generic Programming	[Backhouse et al., 1999]
ADTs and Program Transformation	[Malcolm, 1990]
Law and Order in Algorithmics	[Fokkinga, 1992]
Functional Programming with Morphisms	[Meijer et al., 1991]

9.2 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. Since each generic programming library comes with some boilerplate code, for example for generating embedding-projection pairs, we expect that generic programming libraries will be accompanied by code-generation tools.

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 Portuguese Foundation for Science and Technology (FCT), via the SFRH/BD/35999/2007 grant.

We are grateful to many people for their comments on these lecture notes. The anonymous referee suggested many improvements to the text. 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.

References

Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)

Alimarine, A., Plasmijer, R.: A generic programming extension for Clean. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 168–186. Springer, Heidelberg (2002)

Alimarine, A., Smetsers, S.: Optimizing generic functions. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 16–31. Springer, Heidelberg (2004)

Altenkirch, T., McBride, C.: Generic programming within dependently typed programming. In: Gibbons, J., Jeuring, J. (eds.) Generic Programming. IFIP, vol. 243, pp. 1–20. Kluwer Academic Publishers, Dordrecht (2003)

Backhouse, R., de Bruin, P., Malcolm, G., Voermans, E., van der Woude, J.: Relational catamorphisms. In: Möller, B. (ed.) Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs, pp. 287–318. Elsevier Science Publishers B.V, Amsterdam (1991)

Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming—an introduction. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 28–115. Springer, Heidelberg (1999)

Benke, M., Dybjer, P., Jansson, P.: Universes for generic programs and proofs in Sdependent type theory. Nordic Journal of Computing 10(4), 265–289 (2003)

Bernardy, J.-P., Jansson, P., Zalewski, M., Schupp, S., Priesnitz, A.: A comparison of C++ concepts and Haskell type classes. In: ACM SIGPLAN Workshop on Generic Programming. ACM, New York (2008)

Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. In: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. EATCS (2004), ISBN 3-540-20854-2

Bird, R., Meertens, L.: Nested datatypes. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 52–67. Springer, Heidelberg (1998)

Bird, R., de Moor, O.: Algebra of programming. Prentice-Hall, Englewood Cliffs (1997)

Cheney, J., Hinze, R.: A lightweight implementation of generics and dynamics. In: Chakravarty, M. (ed.) Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, Haskell 2002, pp. 90–104. ACM, New York (2002)

Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York (2000)

Clavel, M., Durán, F., Martí-Oliet, N.: Polytypic programming in Maude. Electronic Notes in Theoretical Computer Science, vol. 36, pp. 339–360 (2000)

Dornan, C., Jones, I., Marlow, S.: Alex User Guide (2003), `http://www.haskell.org/alex`

Fokkinga, M.M.: Law and Order in Algorithmics, PhD thesis. University of Twente (1992)

Forman, I.R., Danforth, S.H.: Putting metaclasses to work: a new dimension in object-oriented programming. Addison Wesley Longman Publishing Co., Inc., Redwood City (1999)

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An extended comparative study of language support for generic programming. Journal of Functional Programming 17(2), 145–205 (2007)

Gibbons, J.: Datatype-generic programming. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 1–71. Springer, Heidelberg (2007)

The Haskell Prime list. Haskell prime (2006), `http://hackage.haskell.org/trac/haskell-prime`

Hinze, R.: Generic programs and proofs. Bonn University, Habilitation (2000)

Hinze, R.: Generics for the masses. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, pp. 236–243. ACM, New York (2004)

Hinze, R.: Generics for the masses. Journal of Functional Programming 16, 451–482 (2006)

Hinze, R., Jeuring, J.: Generic Haskell: applications. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 57–96. Springer, Heidelberg (2003)

Hinze, R., Jeuring, J.: Generic Haskell: practice and theory. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming. LNCS, vol. 2793, pp. 1–56. Springer, Heidelberg (2003)

Hinze, R., Löh, A.: Generic programming in 3D. Science of Computer Programming (to appear, 2009)
Hinze, R., Löh, A.: "Scrap Your Boilerplate" revolutions. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 180–208. Springer, Heidelberg (2006)
Hinze, R., Löh, A.: Generic programming, now! In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 150–208. Springer, Heidelberg (2007)
Hinze, R., Löh, A., Oliveira, B.C.d.S.: "Scrap Your Boilerplate" reloaded. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 13–29. Springer, Heidelberg (2006)
Hinze, R., Jeuring, J., Löh, A.: Comparing approaches to generic programming in Haskell. In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 72–149. Springer, Heidelberg (2007)
Holdermans, S., Jeuring, J., Löh, A., Rodriguez, A.: Generic views on data types. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 209–234. Springer, Heidelberg (2006)
Jeuring, J., Leather, S., Magalhães, J.P., Yakushev, A.R.: Libraries for generic programming in Haskell. Technical Report UU-CS-2008-025, Department of Information and Computing Sciences. Utrecht University (2008)
Karvonen, V.A.J.: Generics for the working ML'er. In: Proceedings of the 2007 Workshop on ML, ML 2007, pp. 71–82. ACM, New York (2007)
Kiselyov, O.: Smash your boilerplate without class and typeable (2006), `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`
Kiselyov, O.: Compositional gmap in SYB1 (2008), `http://www.haskell.org/pipermail/generics/2008-July/000362.html`
Lämmel, R., Jones, S.P.: 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, pp. 26–37. ACM, New York (2003)
Lämmel, R., Jones, S.P.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, pp. 244–255. ACM, New York (2004)
Lämmel, R., Jones, S.P.: Scrap your boilerplate with class: extensible generic functions. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, pp. 204–215. ACM, New York (2005)
Lempsink, E., Leather, S., Löh, A.: Type-safe diff for families of datatypes (submitted for publication, 2009)
Lodder, J., Jeuring, J., Passier, H.: An interactive tool for manipulating logical formulae. In: Manzano, M., Pérez Lancho, B., Gil, A. (eds.) Proceedings of the Second International Congress on Tools for Teaching Logic (2006)
Löh, A.: Exploring Generic Haskell. PhD thesis, Utrecht University (2004)
Löh, A., Hinze, R.: Open data types and open functions. In: Maher, M. (ed.) Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming, PPDP 2006, pp. 133–144. ACM, New York (2006)
Malcolm, G.: Algebraic data types and program transformation. PhD thesis, Department of Computing Science. Groningen University (1990)
Marlow, S., Gill, A.: Happy User Guide (1997), `http://www.haskell.org/happy`
Meertens, L.: Calculate polytypically! In: Kuchen, H., Swierstra, S.D. (eds.) PLILP 1996. LNCS, vol. 1140, pp. 1–16. Springer, Heidelberg (1996)
Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)
Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)

Mitchell, N., Runciman, C.: Uniform boilerplate and list processing. In: Proceedings of the 2007 ACM SIGPLAN workshop on Haskell, Haskell 2007. ACM, New York (2007)

Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Göteborg University (2007)

Norell, U., Jansson, P.: Polytypic programming in Haskell. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, pp. 168–184. Springer, Heidelberg (2004)

Norell, U., Jansson, P.: Prototyping generic programming in Template Haskell. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 314–333. Springer, Heidelberg (2004)

Bruno C. d. S. Oliveira, Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Nilsson, H. (ed.) Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, vol. 7, pp. 199–216 (2006)

Palsberg, J., Barry Jay, C.: The essence of the visitor pattern. In: Proceedings of the 22nd IEEE Conference on International Computer Software and Applications, COMPSAC 1998, pp. 9–15 (1998)

Passier, H., Jeuring, J.: Feedback in an interactive equation solver. In: Seppälä, M., Xambo, S., Caprotti, O. (eds.) Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006, pp. 53–68. Oy WebALT Inc. (2006)

Pfeifer, H., Ruess, H.: Polytypic proof construction. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 55–72. Springer, Heidelberg (1999)

Reinke, C.: Traversable functor data, or: X marks the spot (2008), `http://www.haskell.org/pipermail/generics/2008-June/000343.html`

Rodriguez, A.: Towards Getting Generic Programming Ready for Prime Time. PhD thesis, Utrecht University (2009)

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.C.d.S.: Comparing libraries for generic programming in haskell. Technical report, Utrecht University (2008a)

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., Oliveira, B.C.d.S.: Comparing libraries for generic programming in haskell. In: Haskell Symposium 2008 (2008b)

Rodriguez, A., Holdermans, S., Löh, A., Jeuring, J.: Generic programming with fixed points for mutually recursive datatypes. In: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2009 (2009)

Sheard, T.: Using MetaML: A staged programming language. In: Revised Lectures of the Third International School on Advanced Functional Programming (1999)

Sheard, T., Jones, S.P.: Template metaprogramming for Haskell. In: Chakravarty, M.M.T. (ed.) ACM SIGPLAN Haskell Workshop 2002, pp. 1–16. ACM, New York (2002)

Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1999)

Universiteit Utrecht. EMGM (2008), `http://www.cs.uu.nl/wiki/GenericProgramming/EMGM`

Wadler, P.: Comprehending monads. In: Proceedings of the 1990 ACM conference on LISP and Functional Programming, LFP 1990, pp. 61–78. ACM, New York (1990)

Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, pp. 60–76. ACM, New York (1989)

Weirich, S.: RepLib: a library for derivable type classes. In: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, Haskell 2006, pp. 1–12. ACM, New York (2006)

Yallop, J.: Practical generic programming in OCaml. In: Proceedings of the 2007 workshop on Workshop on ML, ML 2007, pp. 83–94. ACM, New York (2007)