

Comparing generic programming libraries

Alex Gerdes

Master's thesis
Department of Computer Science
Open University

October 2007

Graduation commission: prof. dr. J.T. Jeurig *Professor OU*
dr. B.J. Heeren *Lecturer OU*

Student number: 838147092

Conducted at: Utrecht University

*One Library to rule them all, One Library to join them, One Library
to bring them all, and in the community bind them.*

Adapted quote out of Lord of the Rings (LOTR) from J.R.R. Tolkien.

Abstract

There are numerous approaches to generic programming in Haskell. Until now there hasn't emerged a clear winner. The time is right to join all the scattered effort and to converge into one *common generic programming library*. This master's thesis takes the first step in that process and develops a set of criteria that can be used to evaluate and compare the known generic programming libraries in Haskell. Besides the criteria, a set of test functions has been developed to test whether a generic programming library satisfies a particular criterion. A first preliminary evaluation of a few libraries is conducted. This master's thesis paves the way for the design of a common generic programming library in Haskell.

Contents

| | |
|--|-------------|
| Abstract | v |
| Preface | xi |
| Summary | xiii |
| Samenvatting | xv |
| 1 Introduction | 1 |
| I Prerequisites | 3 |
| 2 Background information | 5 |
| 2.1 Functional programming | 5 |
| 2.2 Theoretical concepts | 7 |
| 2.3 Haskell | 8 |
| 2.3.1 Type classes | 8 |
| 2.3.2 Algebraic data types | 10 |
| 2.4 Data types | 12 |
| 3 Advanced language features | 15 |
| 3.1 GADTs | 15 |
| 3.2 Monads | 17 |
| 3.3 Existential quantification | 19 |
| 3.4 Arbitrary rank polymorphism | 21 |
| 3.5 Multi-parameter type classes | 21 |
| 3.6 Instance declarations | 22 |
| 3.7 Deriving | 24 |
| 3.8 Template Haskell | 24 |
| 4 Generic programming | 25 |
| 4.1 Type-indexed functions | 26 |
| 4.2 Generic algorithms | 28 |

CONTENTS

| | | |
|-----------|--|-----------|
| 4.3 | Generic functions | 30 |
| II | Criteria | 35 |
| 5 | Research Goal | 37 |
| 5.1 | Problem description | 37 |
| 5.2 | Goal | 38 |
| 5.3 | Research approach | 39 |
| 6 | Generic programming libraries | 43 |
| 6.1 | Light-weight approaches | 45 |
| 6.1.1 | Derivable Type Classes | 45 |
| 6.1.2 | Extensible and Modular Generics for the Masses . . . | 46 |
| 6.1.3 | Generic Programming, Now! | 47 |
| 6.1.4 | LIGD | 49 |
| 6.1.5 | Light-weight PolyP | 51 |
| 6.1.6 | RepLib | 51 |
| 6.2 | Strategic Programming | 53 |
| 6.2.1 | Scrap Your Boilerplate | 54 |
| 6.2.2 | Smash Your Boilerplate | 55 |
| 6.2.3 | SYB Revolutions | 56 |
| 6.2.4 | Uniplate | 57 |
| 7 | Criteria | 59 |
| 7.1 | Organisation | 60 |
| 7.2 | Details | 60 |
| 7.2.1 | Types | 61 |
| 7.2.2 | Expressiveness | 63 |
| 7.2.3 | Usability | 65 |
| 8 | Test functions | 67 |
| 8.1 | Equality | 67 |
| 8.1.1 | Generic Rose Trees | 69 |
| 8.1.2 | Generic nested rose trees | 69 |
| 8.1.3 | Trees with weights | 69 |
| 8.2 | Reduce | 70 |
| 8.2.1 | Nested data types | 71 |
| 8.3 | Generic Map | 71 |
| 8.3.1 | GMapQ | 72 |
| 8.4 | FoldInt | 72 |
| 8.5 | Generic Show | 73 |
| 8.5.1 | GADT | 74 |
| 8.6 | Increase | 74 |

CONTENTS

| | | |
|-----------------------|---------------------------------------|-----------|
| 8.7 | Generic transpose | 75 |
| 8.8 | Performance test | 75 |
| 8.9 | Generic lookup | 76 |
| 8.10 | Fulltree | 76 |
| 8.11 | Test suite | 77 |
| 8.12 | Criteria coverage | 78 |
| III Evaluation | | 79 |
| 9 | Evaluation | 81 |
| 9.1 | EMGM | 82 |
| 9.2 | SYB Revolutions | 84 |
| 9.3 | Summary remaining libraries | 86 |
| 9.4 | Performance | 87 |
| 10 | Epilogue | 91 |
| 10.1 | Future work | 91 |
| Glossary | | 97 |

Preface

This thesis is the main result of my graduation at the Open University. The research is carried out at the Software Technology center of the Department of Information and Computing Sciences of the Utrecht University. Prof. dr. Johan Jeuring supervised the research.

I first came across functional programming during a course called ‘Concepts of Programming Languages’. I was immediately impressed by the expressive power and elegance of functional programming. The aforementioned course became one of my favourites of the entire curriculum. I was delighted with the opportunity, given by prof. dr. Johan Jeuring, to do my thesis project with functional generic programming as its main topic.

It has been a choice that I have not regret for the tiniest moment imaginable.

Organisation

The thesis assumes no a priori knowledge about generic programming. It is accessible for everyone who is interested in the issues, although some knowledge about software technology will be helpful. Readers familiar with functional programming can safely skip chapters 2 and 3. Those chapters can be used as a reference for definitions used throughout the thesis. Chapter 4 can be skipped for readers with knowledge about generic programming.

Acknowledgements

A more condensed presentation of the material covered in this thesis will be given in the paper *Comparing Libraries for Generic Programming in Haskell* [24], which is currently under development.

Many people have assisted me in some way along the journey with this thesis as its final stop: I am very grateful to Johan Jeuring who gave me the opportunity to do this research and for his excellent supervision and feedback. I have learned a lot since the start of the project. I would like to thank Alexey Rodriguez Yakushev very much for his help, advise and

Preface

the fruitful discussions we frequently had. Another word of thanks goes to Bastiaan Heeren for reviewing this thesis.

Another person I'd like to thank is Dion Kant who has been an inspiring colleague at ASTRON (Netherlands Foundation for Radio Astronomy), and is a good friend.

Finally, I would like to thank my family and friends for their understanding, *time* and support during the long years of study. Especially I would like to express my deepest feelings of gratitude to my wife Anita and my children Mats and Lise — they make it all worthwhile.

Emmen, Oktober 2007

Alex Gerdes

Summary

Tired of writing so-called ‘boilerplate’ code over and over again? Code that follows the same pattern for every instance of a certain algorithm, but is not essential. Generic programming offers a solution for this problem. Generic programming in the functional programming language Haskell is what this thesis is about.

Generic programming allows the programmer to abstract over the *structure* of a type. It is possible to develop functions that work for a range of types, even on ones that have not yet come to pass. Generic programming makes it possible to solve a class of problems once and for all, instead of writing new code for every single instance. An advantage resulting therefrom is that code becomes more stable and more reusable.

Over the last decade generic programming has made a lot of progress. There are several, the last couple of years more than ten, libraries to generic programming using Haskell. Although there is a lot of activity, ‘real-life’ projects hesitate to incorporate generic programming. Presumably due to the fact that choosing one of the various approaches is risky. It is difficult to be sure whether the right choice has been made. Moreover, many libraries are still residing in a development phase.

The risk in using generic programming can be reduced by designing a *common generic programming library* for which continuing support is guaranteed. Continuing support and consensus about the library can only be achieved by developing this library in an international committee. The first step towards a common generic programming language is to evaluate the existing libraries. This research attempts to answer the following research questions:

- What are the analysis criteria?
- How do the different libraries score on these criteria?

The development of the common library is beyond the scope of this research. This thesis’ main focus is on the definition of a set of relevant criteria and an evaluation procedure (test suite) to asses them. A limited set of libraries have been evaluated with respect to the criteria.

Summary

The result of this research is an extensive set of well defined criteria. Next to the criteria a test suite has been developed that can be used to evaluate generic programming libraries.

Samenvatting

Ook zo moe van het schrijven van zogenaamde ‘boilerplate’ code, code die voor vele instanties van een algoritme gelijk is maar niets essentieels bijdraagt? Generiek programmeren biedt een oplossing voor dat probleem. Dit document gaat in op generiek programmeren toegespits op de functionele programmeertaal Haskell.

Generiek programmeren maakt het mogelijk te abstraheren over de *structuur* van een type. Het is mogelijk functies te definiëren die voor een hele reeks types werken, zelf voor types die nog niet bestaan en mogelijk in de toekomst worden gedefinieerd. Generiek programmeren staat toe om een bepaald soort probleem voor eens en altijd op te lossen. Een voordeel dat hierdoor ontstaat is dat code beter herbruikbaar en stabiel wordt.

Generiek programmeren heeft veel vooruitgang geboekt in de laatste tien jaar. Er zijn vele, meer dan tien, bibliotheken ontwikkeld die een vorm van generiek programmeren aanbieden in Haskell. Ondanks deze ontwikkelingen, schort het aan ‘real-life’ projecten die generiek programmeren opnemen in hun ontwikkelmogelijkheden. Dit komt waarschijnlijk door het feit dat een keuze maken tussen de verschillende bibliotheken een risico inhoudt. Het is niet duidelijk welke bibliotheek de juiste is en veel bibliotheken bevinden zich nog in een ‘alpha’-fase.

Het risico van het gebruiken van een generiek programmeerbibliotheek kan worden gereduceerd door een *gemeenschappelijke* bibliotheek te ontwikkelen, waarvoor een breed draagvlak geldt en gecontinueerde ondersteuning wordt gegarandeerd. Het ontwikkelen en definiëren van een dergelijke bibliotheek dient in een internationale commissie plaats te vinden om consensus en voortdurende ondersteuning te kunnen garanderen. De eerste stap in de richting van een gemeenschappelijke programmeerbibliotheek is het evalueren en vergelijken van bestaande bibliotheken. Hiertoe heeft dit onderzoek op onderstaande vragen een antwoord gepoogd te vinden:

- Wat zijn de analyse criteria?
- Hoe scoren de afzonderlijke bibliotheken op deze criteria?

Het ontwerpen en ontwikkelen van de bibliotheek zelf valt buiten de bereik van het onderzoek. Er is geconcentreerd op het opstellen van een lijst

Samenvatting

met relevante criteria en een evaluatieprocedure. Tevens zijn er een aantal bibliotheken geëvalueerd.

Het resultaat van het onderzoek is een uitvoerige verzameling van goed omschreven criteria. Tevens is er een testomgeving ontwikkeld waarin generiek programmeerbibliotheken kunnen worden geëvalueerd en vergeleken. Met in achtneming van de criteria en testomgeving zijn een aantal bibliotheken onder de loep genomen.

Chapter 1

Introduction

An imperative programmer who first comes across functional programming (FP) is probably stunned by its expressive power and elegance. Haskell is one of the most popular functional programming languages. It is a general purpose, purely functional programming language. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic data types, pattern matching, list comprehensions, a module system, a monadic Input/Output (I/O) system, and a rich set of primitive data types. There is a popular interpreter (called Hugs) and many compilers of which Glasgow Haskell Compiler (GHC) is the most widespread. More information about the Haskell language can be found in the Haskell language report [26].

Generic programming (GP) allows the programmer to abstract over the *structure* of a type. It is possible to develop functions that work for a range of types, even on ones that have not yet come to pass. Generic programming makes it possible to solve a class of problems once and for all, instead of writing new code for every single instance. This fact increases re-usability and reliability, the latter due to the fact that a function has already been used and tested on several types. Generic programming challenges programmers to find ‘generic’ components, strip them of irrelevant detail and write a generic function.

Over the last decade generic programming has made a lot of progress. There are several, the last couple of years more than 10, approaches to generic programming using Haskell. Although there is a lot of activity, real-life projects hesitate to incorporate generic programming. Presumably due to the fact that choosing one of the various approaches is risky. It is difficult to be sure whether the right choice has been made. It is desirable to have a *common generic programming library*, that combines the effort of all different approaches and guarantees long term support and stability.

The purpose of this research is to evaluate and compare the various approaches to generic programming in Haskell. This comparison is inspired

1 Introduction

by a previous comparison [13], and will be used to design a common generic programming library for Haskell. The implementation lies beyond the scope of this master's thesis.

The thesis is organised as follows: the three chapters in part I provide the necessary background for the discussion in later chapters. Chapter 2 introduces the functional programming language Haskell. Chapter 3 describes the advanced features (extensions) of Haskell that are used by generic programming libraries. Chapter 4 shows the details behind generic programming.

The second part of the thesis, consisting of chapters 5, 6, 7 and 8, focuses on our main topic: criteria for evaluating generic programming libraries in Haskell. After the main motivation for this research has been given in Chapter 5, Chapter 6 introduces the generic programming libraries. Chapter 7 discusses the criteria in detail followed by Chapter 8 which touches upon the test functions that are used to evaluate the libraries with respect to the criteria.

Finally, in part III, Chapter 9 does a preliminary evaluation of some libraries and Chapter 10 concludes.

Part I

Prerequisites

Chapter 2

Background information

This chapter provides the necessary background information to understand other parts of the thesis. The depth and amount of background is based on the computer science curriculum at the Open University (OU). First, functional programming is introduced along with underlying concepts. Second, the functional programming language Haskell is presented. The numerous examples provided in this document are all written in Haskell. Third, a brief introduction to type classes, which is one of Haskell's beloved features.

Readers who are already familiar with these aspects should probably skip ahead to the next section or part. We can recommend the excellent text book by Hudak [20] and the various tutorials and references listed on the Haskell web site at <http://www.haskell.org/>, for readers interested in more than the brief overview as is given here.

2.1 Functional programming

Functional programming offers a high-level view of programming, giving its users a variety of features which help them to build elegant yet powerful and general programs. Central to functional programming is the idea of a function, which computes a result that depends on the values of its inputs. The elegance of functional programming is a consequence of the way that functions are defined: an equation is used to say what the value of a function is on an arbitrary input.

In a functional programming language it is possible to define new functions. The function can be used afterwards, along with standard functions, in expressions. For instance the ever popular¹ factorial function can be defined. The factorial of a number n (mostly written as $n!$) is the product of the numbers 1 to n , for example $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$. In a functional programming language the definition of the function *fac* could look like:

¹in functional programming examples

2 Background information

```
hamm = 1 : map (2*) hamm # map (3*) hamm # map (5*) hamm
  where xxs@(x : xs) # yys@(y : ys)
        | x ≡ y = x : xs # ys
        | x < y = x : xs # yys
        | x > y = y : xxs # ys
```

Figure 2.1: Hamming numbers implementation

```
fac n = product [1..n].
```

This definition uses the notation for list of numbers between two values and the standard function *product*. The standard function *product* multiplies all elements in a list. The newly defined function can be used by another function, for example the function *n choose k*: the number of ways in which *k* objects can be chosen from a collection of *n* objects. Statistics literature states this number equals:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

This definition can, just as with *fac*, be almost literally defined in Haskell:

```
choose n k = fac n `div` (fac k * fac (n - k)).
```

This example is taken from a functional programming course [8] at Utrecht University (UU).

Hamming numbers. Another widely used example, calculating Hamming numbers, shows the elegance and simplicity of functional programming. A Hamming (or regular) number is a whole number which divides a power of 60. In number theory, these numbers are called 5-smooth, because they can be characterised as having only 2, 3, or 5 as prime factors.

Formally, a Hamming number is an integer of the form $2^i \cdot 3^j \cdot 5^k$, for non-negative integers *i*, *j* and *k*. The first few Hamming numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, 40, 45, 48, 50, 54, 60. Although the Hamming numbers appear dense within the range from 1 to 60, they are quite sparse among the larger integers. This feature is exploited in error correcting applications.

Figure 2.1 shows a function that calculates an infinite list of Hamming numbers. This program is strikingly compact; you can read the algorithm straight off it. Lazy evaluation allows us to define an infinite list in terms of itself. A user-defined lexical operator *#* makes the function more readable. The *#* operator merges the infinite lists together.

The above examples use the syntax of the functional programming language Haskell. Section 2.3 gives more information about Haskell.

2.2 Theoretical concepts

This section describes some theoretical concepts of functional programming. The concepts explain the underlying techniques used in functional programming paradigm. Many features that stem from these concepts are used in generic programming libraries.

Higher-order functions. A function is higher-order when it takes another function as argument. Higher-order functions are closely related to first-class functions, in that higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: ‘higher-order’ describes a mathematical concept of functions that operate on other functions, while ‘first-class’ is a computer science term that describes programming language entities that have no restriction on their use. First-class functions can appear anywhere in the program where other first-class entities like numbers can appear, including as arguments to other functions and as their return values. Higher-order functions enable currying, a technique in which a function is applied to its arguments one at a time, with each application returning a new (higher-order) function that accepts the next argument.

Pure functions. Purely functional programs have no side effects. This makes it easier to reason about their behaviour. For example, the result of applying a pure function to pure arguments does not depend on the order of evaluation. As a result, a language which has no impure functions (a ‘purely functional language’, such as Haskell) may use call-by-need evaluation. However, not all functional languages are pure. The Lisp family of languages are not pure because they allow side-effects. Since pure functions do not modify shared variables, pure functions can be executed in parallel without interfering with one another. Pure functions are therefore thread-safe, which allow interpreters and compilers to use call-by-future evaluation. Pure functional programming languages typically enforce referential transparency, which is the notion that ‘equals can be substituted for equals’: if two expressions have ‘equal’ values (for some notion of equality), then one can be substituted for the other in any larger expression without affecting the result of the computation.

Recursion. Iteration in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over.

Common patterns of recursion can be factored out using higher-order functions, catamorphisms and anamorphisms (or ‘folds’ and ‘unfolds’) being the most obvious examples. Such higher-order functions play a role analogous to built-in control structures such as loops in imperative languages.

2 Background information

Strict, non-strict and lazy evaluation. Functional languages can be categorised by whether they use strict or non-strict evaluation, concepts that state how function arguments are processed when an expression is being evaluated. Under strict evaluation arguments to a function are evaluated before the function call; non-strict evaluation passes arguments to the function unevaluated and the calling function determines when to evaluate the arguments.

Strict evaluation has efficiency advantages. An argument is evaluated once with strict evaluation, while it may be evaluated multiple times with non-strict evaluation. However there are reasons for preferring non-strict evaluation. Non-strict evaluation provides a more expressive language. For example, it supports infinite data structures, such as a list of all Hamming numbers, as we have seen in Section 2.1. Such structures are of use when an unknown but finite part of the structure is required. In that case strict evaluation might calculate too much. The need for a more efficient form of non-strict evaluation led to the development of lazy evaluation, a type of non-strict evaluation, where the initial evaluation of an argument is shared throughout the evaluation sequence. Consequently an argument is never evaluated more than once. Lazy evaluation is used by Haskell.

2.3 Haskell

Haskell is one of the most popular functional programming language. The Haskell language was conceived during a meeting held at the 1987 Functional Programming and Computer Architecture conference. At the time it was believed that the advancement of functional programming was being stifled by the wide variety of languages available. There were more than a dozen lazy, purely functional languages in existence and none had widespread support. A committee was formed to design the language.

The name Haskell was chosen in honour of the mathematician Haskell Curry, whose research forms part of the theoretical basis upon which many functional languages are implemented. Haskell is a lazy functional language with polymorphic higher-order functions, algebraic data types and list comprehensions. It has a module system, and supports ad-hoc polymorphism (via classes). Haskell is purely functional, even for Input/Output (I/O). Most Haskell implementations come with a number of libraries supporting arrays, complex numbers, infinite precision integers, operating system interaction, concurrency and mutable data structures.

2.3.1 Type classes

This section describes the class declarations that are used to introduce new (single parameter) type classes in Haskell and the instance declarations that are used to populate them.

A class declaration specifies the name for a class and lists the member functions that each type in the class is expected to support. The actual types in each class, which are normally referred to as the instances of the class, are described using separate (instance) declarations, as described below. For example the *Eq* class, representing the set of equality types, is defined by the following declaration:

```
class Eq a where
  ( $\equiv$ ) :: a → a → Bool
  ( $\neq$ ) :: a → a → Bool.
```

The type variable *a*, that appears in the class declaration above, represents an arbitrary instance of the class. The intended reading of the declaration is that, if *a* is a particular instance of *Eq*, then we can use the (\equiv) operator at type $a \rightarrow a \rightarrow Bool$ to compare values of type *a*.

The restriction on the use of the equality operator is reflected in the type that is assigned to it:

$$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool.$$

Types that are restricted by a predicate like this are referred to as *qualified types*. Such types will be assigned to any function that makes either direct or indirect use of the member functions of a class at some unspecified type. For example, the functions:

```
member x xs = any (x  $\equiv$ ) xs
subset xs ys = all ( $\lambda$ x → member x ys) xs
```

will be assigned types:

```
member :: Eq a ⇒ a → [a] → Bool
subset :: Eq a ⇒ [a] → [a] → Bool.
```

Classes may be arranged in a hierarchy and may have multiple member functions. The following example illustrates both with a declaration of the *Ord* class, which contains the types whose elements can be ordered using comparison operators:

```
class Eq a ⇒ Ord a where
  (<), (≤) :: a → a → Bool.
```

In this particular context, the \Rightarrow symbol should not be read as implication; the intention is that every instance of *Ord* is also an instance of *Eq*. Thus *Eq* plays the role of a *superclass* of *Ord*. This mechanism allows the programmer to specify an expected relationship between classes: it is the compiler's responsibility to ensure that this property is satisfied, or to produce an error message if it is not.

2 Background information

The instances of any given class are described by a collection of instance declarations. For example, the following declarations show how to define equality for booleans and pairs:

```
instance Eq Bool where  
  x ≡ y = if x then y else not y  
instance (Eq a, Eq b) ⇒ Eq (a, b) where  
  (x, y) ≡ (u, v) = (x ≡ u ∧ y ≡ v).
```

The first line of the second instance declaration tells us that equality on values of type a and b is needed to provide an equality on pairs of type (a, b) . Even with just these two declarations, we have already specified an equality operation on the infinite family of types that can be constructed from *Bool* by repeated uses of pairing. Additional declarations, which may be distributed over many modules, can be used to extend the class to include other data types.

2.3.2 Algebraic data types

In a typed programming language variables have a certain type, which indicates the sort of variable. Examples of primitive types are integers, characters and booleans. A variable with an integer type contains a value that falls into the integer range ($\in \mathbb{Z}$). Haskell checks types at compile-time, so using a boolean variable where a character is expected (e.g. the *toUpper* function that capitalises a character), will result in a type error. The type system guards users against type errors and all the problems that originate from it (e.g. memory violations).

Haskell's type system [26] has a three level structure. The lowest level consists of values, like 1 and 'a'. The second level, the type level, describes the structure of the value. The third level describes the structure of the type, which is called the *kind* of a type. The third level is necessary to allow parametric types, like lists that are inhabited by an arbitrary type. A parametric type can be seen as a function on types and the kind system allows us to specify this in a precise way. For example, the list data type takes a type variable as a parameter and results in a type. This has kind $\star \rightarrow \star$ whereas a non-parametric type has kind \star . A kind can be seen as the 'type' of a type.

In Haskell a new data type can be defined with one or more *constructors*. Constructors can be used much like functions in that they can be (partially) applied to arguments of the appropriate type. A constructor application cannot be reduced (evaluated) like a function application though, since it is already in normal form. Functions which operate on algebraic data types can be defined using pattern matching. The actual data is wrapped in constructors, they are the arguments of a constructor. Special cases of algebraic types are product types (only one constructor) and enumeration

types (many constructors with no arguments). In Haskell algebraic data types are defined with a *data* definition. Here are four examples: the type of complex numbers, pairs, lists and binary trees:

```
data Complex = C Float Float
data Pair a b = P a b
data List a = Empty | Con a (List a)
data BinTree a = Leaf a | Bin (BinTree a) (BinTree a).
```

In general a data type declaration of the schematic form:

```
data T a1 ... as = C1 t1,1 ... t1,m1 | ... | Cn tn,1 ... tn,mn
```

introduces data constructors C_1, \dots, C_n with signatures:

$$C_i :: \forall a_1 \dots a_s. t_{i,1} \rightarrow \dots \rightarrow t_{i,m_i} \rightarrow T a_1 \dots a_s.$$

The list constructors *Empty* and *Con* are written [] and ‘.’ in Haskell. The following alternative definition of the pair data type:

```
data Pair a b = Pair { outl :: a, outr :: b }
```

makes use of Haskell’s *record* syntax: the declaration introduces the data constructor *Pair* and two functions to access pairs:

$$\begin{aligned} outl &:: \forall a b. Pair a b \rightarrow a \\ outr &:: \forall a b. Pair a b \rightarrow b. \end{aligned}$$

Pairs, lists and binary trees are examples of parameterised data types or type constructors. The kind of manifest types such as *Complex* is \star , whereas the kind of a type constructor is a function of the kind of its parameters (type arguments). The kind of *Pair* is $\star \rightarrow \star \rightarrow \star$, the kind of [] as well as *BinTree* is $\star \rightarrow \star$.

Overloading. Member functions of type classes can be overloaded for new data types. For example, the equality function can be overloaded by making an instance declaration of the *Eq* class. As an example, the instance declaration for the *BinTree* type is:

```
instance Eq  $\alpha \Rightarrow$  Eq (BinTree  $\alpha$ ) where
  Leaf l    $\equiv$  Leaf r   = l  $\equiv$  r
  Bin ll lr  $\equiv$  Bin rl rr = ll  $\equiv$  rl  $\wedge$  lr  $\equiv$  rr
  _        $\equiv$  _       = False.
```

The instance declaration has to be provided for every new algebraic data type if the use of the (\equiv) operator is desired. Generic programming solves this problem, the equality function only has to be defined once and works for all new types². More on this in Chapter 4.

²For a limited set of type classes the instances can be derived.

2.4 Data types

The data type construct in Haskell combines many aspects: type abstraction and application, recursion, records, polymorphism, etc. We use the following data types in examples used in this document and in tests for generic libraries (Chapter 8). These examples cover many of the aspects of Haskell data types, but we do not try to be complete. An aspect that is missing is: higher-rank constructors (explicit forall in the data type declaration).

Binary trees. From the previous section: the type *BinTree* is a standard binary tree data type with nodes with two subtrees and values in the leaves:

```
data BinTree a = Leaf a | Bin (BinTree a) (BinTree a).
```

This data type has kind $\star \rightarrow \star$. A binary tree can contain any type of values in the leaves. For example:

```
bintree :: BinTree Int
bintree = Bin (Bin (Leaf 32) (Leaf 3)) (Leaf 4).
```

Company data type. The *Company* data type represents the organisational structure of a company. The *Company* data type uses a set of monomorphic data types. These data types make use of products (*Employee*, *Person*), sums (*DUnit*), another data type such as the type of lists and mutual recursion (*Dept* and *DUnit*). Furthermore, *Name* and *Address* (which are strings and so technically lists in Haskell) would most of the time be treated differently from the list of *Units* or *Depts*. The following definitions declare the compound data type *Company*:

```
data Company = C [Dept]
data Dept    = D Name Manager [DUnit]
data DUnit  = PU Employee | DU Dept
data Employee = E Person Salary
data Person  = P Name Address
data Salary  = S Float
type Manager = Employee
type Name    = String
type Address = String.
```

Trees with weights. We adapt the type of binary trees such that we can assign a weight to a (sub)tree. This data type is like the *BinTree* data type, but it has two type arguments instead of a single type argument. Moreover, there is an extra constructor *WithWeight*, which adds a weight

to a (sub)tree. The weight type may be different from the type of the information stored in the leaves. Even if these types are the same, weights may be treated differently from the leaf values. The definition of *WTree*:

```
data WTree a w = WLeaf a
                | WFork (WTree a w) (WTree a w)
                | WithWeight (WTree a w) w.
```

The data type *WTree* has kind $\star \rightarrow \star \rightarrow \star$.

Rose trees. Rose trees are trees in which internal nodes have a list of children instead of just two:

```
data Rose a = Node a [Rose a].
```

We abstract from the list data type to obtain the data type of generalised rose trees:

```
data GRose f a = GNode a (f (GRose f a)).
```

An internal node may now have a structure of children, where a structure is a data type of kind $\star \rightarrow \star$. The interesting aspect of the *GRose* data type is that it is higher-order kinded: it takes a type constructor as argument, to produce a type constructor $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$.

The following is an example value of type *GRose* `[] Int`:

```
grose = GNode 2 [GNode 1 [], GNode 2 []].
```

Perfect trees. The data type *Perfect* is used to model perfect binary trees: binary trees that have exactly 2^n elements, where n is the depth of the binary tree. The *Perfect* data type is defined as follows:

```
data Perfect a = Zero a | Succ (Perfect (Fork a))
data Fork a    = Fork a a.
```

The depth of a perfect binary tree is the Peano number represented by its constructors. The data type *Perfect* is a so-called *nested data type* [2], because the type argument changes from a to *Fork a* in the recursion.

Generalised Algebraic Data Type. The data type *Expr*, explained in more detail in Section 3.1, represents a small *expression* language. Note that Generalised Algebraic Data Types (GADTs) are not in Haskell98. Here's the definition of *Expr*:

```
data Expr :: * → * where
  Num :: Int → Expr Int
  Plus :: Expr Int → Expr Int → Expr Int
  Eq   :: Expr Int → Expr Int → Expr Bool
  If   :: ∀a. Expr Bool → Expr a → Expr a → Expr a.
```


Chapter 3

Advanced language features

This chapter reviews some extensions of the Glasgow Haskell Compiler. Not all extensions are reviewed, merely those that are used in one (or more) of the generic programming libraries. The online GHC manual [26] is a good place for more information about the extensions.

The extensions presented here are part of GHC's system, they aren't part of Haskell98. In order to use them, you will have to enable the appropriate flag, like `-fglasgow-exts` for existential quantification.

3.1 GADTs

Generalised algebraic data types (GADTs) are a simple but powerful generalisation of data types in Haskell. Recently they have been added to the extensions supported by the Glasgow Haskell Compiler (GHC). The key idea is to allow the type of each data constructor to be stated separately. The *result* type can be different from the declared data type. This extra information can be used when pattern matching on constructors. This allows, for example, that embedded languages can be *statically* type checked.

To demonstrate the power of GADTs, we present an embedded language case. Haskell's data construct can be used to define an embedded language. The following simple expression language is often used in papers about GADTs [27, 43, 46]:

```
data Expr = LInt Int
          | LBool Bool
          | Inc Expr
          | IsZero Expr
          | If Expr Expr Expr.
```

An expression is evaluated to one of the following values¹:

¹The semantics of the expression language

3 Advanced language features

```
data Val = VBool Bool
        | VInt Int deriving Show.
```

The embedded language consists of integer and boolean values, an increment and *IsZero* function and a conditional *If* construct. Now we want to write an evaluator for the given expression language. Here's an attempt:

```
eval :: Expr → Val
eval (LInt n)   = VInt n
eval (Inc e)    = case eval e of
    VInt n → VInt (n + 1)
    _      → error "Inc applied to non-int"
eval (IsZero e) = case eval e of
    VInt n → VBool (n ≡ 0)
    _      → error "IsZero is applied to non-int"
eval (If c e1 e2) = case eval c of
    VBool b → if b then eval e1 else eval e2
    _      → error "condition is non-bool".
```

The *eval* function takes an expression as input and returns its value. The evaluation function does type checking at *run-time* by checking the tags of values, it is so-called *tag-full*. Notice the cases to handle incorrect input, e.g. applying the increment function on a boolean. Incorrect input leads to a run-time error. Ideally we would like to see this error at *compile-time* and the expression language to be *well-typed*.

Next we examine two example evaluations. The evaluation of **if** *IsZero* 1 **then** 1 **else** *Inc* 2 yields:

```
> eval (If (IsZero (LInt 1)) (LInt 1) (Inc (LInt 2)))
VInt 3
```

and the evaluation of *Inc True*:

```
> eval (If (LInt 1) (LInt 1) (Inc (LInt 2)))
*** Exception: condition is non - bool.
```

A trick to make expressions well-typed is to split the data type *Expr*. A downside to this solution is that the evaluation function is also split up and becoming more difficult to implement. It would be a big help if we could specify the return type of a constructor. Well that's what GADTs are all about. The following example tries to overcome the deficiencies discovered previously. Look at the next data declaration:

```
data GExpr a where
  GLInt  :: Int → GExpr Int
  GLBool :: Bool → GExpr Bool
```

$$\begin{aligned}
GIsZero &:: GExpr\ Int \rightarrow GExpr\ Bool \\
GIf &:: GExpr\ Bool \rightarrow GExpr\ a \rightarrow GExpr\ a \rightarrow GExpr\ a.
\end{aligned}$$

It defines a data type with a *type index* a ; the constructors are given type signatures, just like functions. The arguments in the constructor type signature are constructor fields. The constructors may lay restrictions on the type argument in the return type, the desired feature mentioned before. It is now possible to make explicit that $GLInt$ returns a value of type $GExpr\ Int$ instead of $GExpr\ a$.

The constructors use indices to restrict the values that can be plugged in as fields. In this case $GExpr$ represents well-typed expressions.

Using GADTs, the evaluation function becomes easy to implement and beautifully concise:

$$\begin{aligned}
evalG &:: GExpr\ a \rightarrow a \\
evalG\ (GLInt\ i) &= i \\
evalG\ (GLBool\ b) &= b \\
evalG\ (GIsZero\ e) &= evalG\ e \equiv 0 \\
evalG\ (GIf\ c\ e_1\ e_2) &= \mathbf{if}\ evalG\ c\ \mathbf{then}\ evalG\ e_1\ \mathbf{else}\ evalG\ e_2.
\end{aligned}$$

Here's how typing works: the $GExpr\ a$ in the signature is refined to the constructor return type in every arm. So, for $GIsZero$ the right hand side must return $GExpr\ Bool$ rather than $GExpr\ a$. A downside is that the type checking algorithm requires a type signature for the function, annotating is mandatory.

When having a closer look at the type $GExpr\ a$, it might seem that it is a bit unusual. Though $GExpr$ is parameterised, it is not a container type: an element of $GExpr\ Int$, for instance, is an expression that evaluates to an integer; it is not necessarily a data structure that contains integers. This means, that we cannot define a mapping function $(a \rightarrow b) \rightarrow (GExpr\ a \rightarrow GExpr\ b)$ as for many other data types. It is impossible to convert expressions of type $GExpr\ a$ into expression of type $GExpr\ b$. The type $GExpr\ b$ might not even be inhabited: there are, for instance, no expressions of type $GExpr\ String$. This is the reason why $GExpr\ a$ is also called a *phantom type* [5, 10], since the type argument of $GExpr$ is not related to any component.

Generalised algebraic data types enhance the expressiveness of Haskell. It's not all sunshine, a downside is that the types of functions using GADTs cannot be inferred by the compiler and have to be annotated.

3.2 Monads

The only problem with the pure function concept (Section 2.2) is that some actions violate it. For example, input functions and random-number generators don't always return the same value for a given set of arguments.

3 Advanced language features

Output functions, on the other hand, have an effect on the outside world, which is independent of the function's return value. It is much harder for a compiler to optimise the program, is one of the consequences of these 'impure' actions.

Monads are used to alleviate this problem. Some monads, like the I/O monad, are a sort of wrapper or label that mark its content as impure. The compiler separates the results of impure actions from the rest of the program by requiring the programmer to handle all monads in special monad functions. This allows all pure, non-monadic code to be fully optimised. There are many different kinds of monads, each with its own unique properties. For example, the I/O monad handles input and output, the *Maybe* monad handles possible failures and the *List* monad handles multiple results.

Since I/O functions interact with the outside world, which cannot be optimised by the compiler, they can't be called from other kinds of functions. An I/O function can only be called by another I/O function. This means the entry point of every pure functional program has to be an I/O function. In Haskell, this function is *main*; *main* initiates the chain of I/O function calls, pure functions can be called from anywhere in this chain.

How monad functions work. A monad function is a function that returns a monad. It can also take monad arguments. Inside the monad function, there is an operator that lets you pull values out of the monad and safely pass the values to pure functions. You can think of the monad function as removing the contents of one or more monads, operating on the contents functionally and putting the results back into a monad to be returned as the value of the function. Every monad has two basic functions, the $\gg=$ operator, called 'bind', and *return*. The function *return* puts a value back into a monad. The bind operator $\gg=$ combines two monadic functions in a specific way, according to the definition of that particular monad. In the I/O monad, $x \gg= f$ pulls the contents out of the action value x , and passes the raw data to the I/O function f .

The effect of $\gg=$ in the I/O monad is to produce simple sequencing of operations, which is the characteristic quality of I/O. In other monads, the bind operator behaves differently. Each monad has its own particular way, implemented through $\gg=$, of combining operations to produce larger operations.

In addition to the bind operator, every monad has the \gg operator. This operator does the same thing as $\gg=$, except it ignores the value of its first argument.

Syntax. There are two styles of monad syntax in Haskell. The original syntax uses the monad operators directly. A typical I/O monad function might look something like this:

3.3 Existential quantification

```
getName =
  putStr "Please enter your name: " >>
  getLine >>=
  (\name → putStrLn ("Welcome, " ++ name)) >> return name).
```

The I/O function `putStr` prints its text argument to the user's screen. The `>>` operator ignores the return value of `putStr` and calls the next function. The I/O function `getLine` returns an I/O monad containing a string (the user's input). The `>>=` operator pulls the string out of the monad returned by `getLine` and passes it to a nameless (lambda) function that takes a string (name). This function calls the I/O function `putStrLn`, which prints a greeting and passes control to `return`, which returns the string to its monad. This monad is the return value of `getName`.

For many purposes, it is more convenient to use **do** notation, which is based on the original bind operator. The monad function begins with the keyword **do** and contains actions separated by either semicolons or line breaks. A left arrow binds the identifier on the left to the contents of the monad on the right. For example:

```
getName = do
  putStr "Please enter your name: "
  name ← getLine
  putStrLn ("Welcome, " ++ name)
  return name.
```

Notice the above notation is only syntactic sugar and can be defined using the original monad operators.

3.3 Existential quantification

Existential types, or existentials for short, are a way of squashing a group of types into one, single type. The `forall` keyword is used to explicitly bring type variables into scope. For example, consider something you probably have seen before:

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b].$$

In this example a and b are *type variables*, the function works for *all* types a and b . Another way of putting this is that those variables are (implicitly) universally quantified. Quantifiers stem from formal logic: 'for all' (or \forall) and 'exists' (or \exists). They quantify whatever comes after them: $\exists x.P$ (where P is an assertion, like $x > 5$) means that there is at least one x such that P . In contrast to $\forall x.P$, which means that for every x you can prove P .

The `forall` keyword quantifies types in a similar way. We would rewrite the type of `map` as follows:

3 Advanced language features

$$\text{map} :: \forall a b. (a \rightarrow b) \rightarrow [a] \rightarrow [b].$$

The `forall` can be seen to be bringing the type variables a and b into scope. In Haskell, the type of a function implicitly begins with a `forall` keyword, so the two type declarations for `map` are equivalent.

This default behaviour can be overridden explicitly by telling Haskell where the `forall` keyword goes. One use of this is for building existentially quantified types. For example:

$$\text{data } T = \forall a. \text{MkT } a.$$

This means that the constructor `MkT` has type:

$$\text{MkT} :: \forall a. a \rightarrow T.$$

So we can pass any type we want to `MkT` and it will convert it into a T . So what happens when we de-construct a `MkT` value?

$$\text{foo } (\text{MkT } x) = \dots \quad \text{-- what is the type of } x?$$

As we have just stated, x could be of any type. That means it's a member of some arbitrary type, so has the type $x :: \exists a. a$. In other words, our declaration for T is isomorphic to the following one:

$$\text{data } T = \text{MkT } (\exists a. a) \quad \text{-- pseudo-Haskell ..}$$

In this way we have constructed an existential type. We can use this to make, for instance, a heterogeneous list:

$$\text{heteroList} = [\text{MkT } 5, \text{MkT } (), \text{MkT } \text{True}, \text{MkT } \text{map}].$$

Of course, when we pattern match on `heteroList` we can't do anything with its elements, as all we know is that they have some arbitrary type. However, if we are to introduce class constraints:

$$\text{data } T' = \forall a. \text{Show } a \Rightarrow \text{MkT}' a$$

which is isomorphic to:

$$\text{data } T' = \text{MkT}' (\exists a. \text{Show } a \Rightarrow a).$$

Then we are able to use its elements. The class constraint serves to limit the types we're unioning over, so that now we know the values inside a `MkT'` are elements of some arbitrary type which instantiates `Show`. The implication of this is that we can apply `show` to a value of type $\exists a. \text{Show } a \Rightarrow a$. It doesn't matter exactly which type it turns out to be.

3.4 Arbitrary rank polymorphism

Haskell type signatures are implicitly (universally) quantified. The keyword `forall` allows us to say exactly what this means. For example:

$$f :: a \rightarrow a$$

means this:

$$f :: \forall a. (a \rightarrow a).$$

The two are treated identically, as we have seen before in Section 3.3. However, GHC's type system supports arbitrary-rank explicit universal quantification in types. For example, all the following types are legal:

$$\begin{aligned} f_1 &:: \forall a b. a \rightarrow b \rightarrow a \\ g_1 &:: \forall a b. (Ord\ a, Eq\ b) \Rightarrow a \rightarrow b \rightarrow a \end{aligned}$$

$$\begin{aligned} f_2 &:: (\forall a. a \rightarrow a) \rightarrow Int \rightarrow Int \\ g_2 &:: (\forall a. Eq\ a \Rightarrow [a] \rightarrow a \rightarrow Bool) \rightarrow Int \rightarrow Int \end{aligned}$$

$$f_3 :: ((\forall a. a \rightarrow a) \rightarrow Int) \rightarrow Bool \rightarrow Bool.$$

Here, f_1 and g_1 are rank-1 types and can be written in Haskell98 (e.g. $f_1 :: a \rightarrow b \rightarrow a$). The `forall` makes explicit the universal quantification that is implicitly added by Haskell.

The functions f_2 and g_2 have rank-2 types; the `forall` is on the left of a function arrow. As g_2 shows, the polymorphic type on the left of the function arrow can be overloaded. The function f_3 has a rank-3 type; it has rank-2 types on the left of a function arrow. GHC allows types of arbitrary rank; you can nest `forall`s arbitrarily deep in function arrows.

3.5 Multi-parameter type classes

Multi-parameter type classes allow for multiple class parameters. For example:

```
class Collects e ce where
  empty  :: ce
  insert :: e -> ce -> ce
  member :: e -> ce -> Bool.
```

This is one of the most commonly suggested applications for multiple parameter type classes. It provides uniform interfaces to a wide range of collection types. Such types might be expected to offer ways to construct empty collections, to insert values, to test for membership, and so on.

3 Advanced language features

The type variable e used here represents the element type, while ce is the type of the collection itself. Within this framework, we might want to define instances of this class for lists or characteristic functions. Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e  $\Rightarrow$  Collects e [e] where {}  
instance Eq e  $\Rightarrow$  Collects e (e  $\rightarrow$  Bool) where {}.
```

All this looks quite promising; we have a class and a range of interesting implementations. Unfortunately, there are some serious problems with the class declaration. First, the empty function has an ambiguous type:

```
empty :: Collects e ce  $\Rightarrow$  ce.
```

By ‘ambiguous’ we mean that there is a type variable e that appears on the left of the \Rightarrow symbol, but not on the right. The problem with this is that, according to the theoretical foundations of Haskell overloading, we cannot guarantee a well defined semantics for any term with an ambiguous type. For this reason, Haskell will reject any attempt to define or use such terms.

This can be resolved by another feature, functional dependencies [25] to retain unambiguity. The functional dependency $ce \rightarrow e$ states that for all instance declarations of *Collects* the element type can be uniquely determined from the collection type. In this case *empty* is unambiguous. The functional dependency is placed after the type variables:

```
class Collection e ce | ce  $\rightarrow$  e where {}.
```

3.6 Instance declarations

An instance declaration has the form:

```
instance (a1, ..., an)  $\Rightarrow$  C t1 ... tm where {}.
```

The part before the \Rightarrow is the context, while the part after the \Rightarrow is the head of the instance declaration. In Haskell98 the head of an instance declaration must be of the form $C (T a_1 \dots a_n)$, where C is the class, T is a type constructor (and not a type synonym) and the $a_1 \dots a_n$ are distinct type variables. Furthermore, the assertions in the context of the instance declaration must be of the form $C a$ where a is a type variable that occurs in the head.

Examples are given throughout the document, for instance in the text on type classes in Section 2.3.1.

Undecidable instances. Sometimes you might want to use the following to get the effect of a ‘class synonym’:

```
class (C1 a, C2 a, C3 a) ⇒ C a
```

```
instance (C1 a, C2 a, C3 a) ⇒ C a.
```

This allows you to write shorter signatures:

```
f :: C a ⇒ { }
```

instead of:

```
f :: (C1 a, C2 a, C3 a) ⇒ { }.
```

For this kind of behaviour, you need to enable the `-fundecidable-instances` flag.

Overlapping and incoherent instances. In general, GHC requires that it is unambiguous which instance declaration should be used to resolve a type class constraint. This behaviour can be modified by two flags: `-fallow-overlapping-instances` and `-fallow-incoherent-instances`.

When resolving the constraint `C Int Bool`, it tries to match every instance declaration against the constraint, by instantiating the head of the instance declaration. For example, consider these declarations²:

```
instance ctx1 ⇒ C Int a   where ... -- (A)
instance ctx2 ⇒ C a Bool where ... -- (B)
instance ctx3 ⇒ C Int [a] where ... -- (C)
instance ctx4 ⇒ C Int [Int] where ... -- (D).
```

The instances (A) and (B) match the constraint `C Int Bool`, but (C) and (D) do not. When matching, the compiler (GHC) takes no account of the context of the instance declaration (`ctx1` etc). The default behaviour is that exactly one instance must match the constraint it is trying to resolve. It is fine if there is a potential of overlap (by including both declarations (A) and (B)); an error is only reported if a particular constraint matches more than one.

The `-fallow-overlapping-instances` flag instructs the compiler to allow more than one instance to match, provided there is a most specific one. For example, the constraint `C Int [Int]` matches instances (A), (C) and (D), but the last is more specific, and hence is chosen. If there is no most-specific match, the program is rejected.

²These examples are taken from the GHC system user’s guide [47].

3 Advanced language features

However, the compiler is conservative about committing to an overlapping instance. For example:

$$\begin{aligned} f &:: [b] \rightarrow [b] \\ f\ x &= \{ \}. \end{aligned}$$

Suppose that from the right hand side of f we get the constraint $C\ Int\ [b]$. However, the compiler does not commit to instance (C), because in a particular call of f , b might be instantiated to Int , in which case instance (D) would be more specific. The program will be rejected. If you add the flag `-fallow-incoherent-instances`, (C) will be picked instead, without complaining about the problem of subsequent instantiations.

The `incoherent`-flag implies the `overlapping`-flag, but not vice versa.

3.7 Deriving

Haskell98 allows the programmer to add **deriving** (Eq, Ord) to a data type declaration, to generate a standard instance declaration for classes specified in the deriving clause. In Haskell98, the only classes that may appear in the deriving clause are the standard classes $Eq, Ord, Enum, Ix, Bounded, Read$ and $Show$.

GHC extends this list with two more classes that may be automatically derived (provided the `-fglasgow-exts` flag is specified): $Typeable$, and $Data$. These classes are defined in the library modules $Data.Typeable$ and $Data.Generics$ respectively and the appropriate class must be in scope before it can be mentioned in the deriving clause.

An instance of $Typeable$ can only be derived if the data type has seven or fewer type parameters, all of kind \star .

3.8 Template Haskell

Template Haskell (TH) is an extension to Haskell98 that allows you to do type-safe compile-time *meta-programming*, with Haskell both as the manipulating language and the language being manipulated.

Template Haskell provides new language features that allow us to convert back and forth between concrete syntax, i.e. what you would type when you write normal Haskell code, and abstract syntax trees. These abstract syntax trees are represented using Haskell data types and, at compile time, they can be manipulated by Haskell code. This allows you to reify (convert from concrete syntax to an abstract syntax tree) some code, transform it and splice it back in (convert back again), or even to produce completely new code and splice that in, while the compiler is compiling your module.

The `-fth` flag enables these features.

Chapter 4

Generic programming

In software technology it is desired that ‘programming patterns’ can be named and reused. This has been a driving force in the development of higher-order programming languages. To which extent a ‘programming pattern’ (entities) can be used, determines the *genericity* of a programming language. The level of *genericity* can be indicated by the following questions:

- Which entities can be named in a definition and then referred to by that name?
- Which entities can be supplied as parameters?
- Which entities can be used ‘anonymously’, in the form of an expression, as parameters?

An entity for which all three are possible is called *first-class citizen* of that language. For instance a function is a first-class citizen of Haskell. Functional programming languages excel in the evolution of programming languages because of the high-level of abstraction that is achieved by the combination of higher-order functions and parametric polymorphism. However, the level of genericity still has its limitations. Types can be defined *and* used as parameters, but only in ‘type expressions’. They cannot be passed to functions. This is where generic programming comes in [1].

Designing a data type, to which functionality is added, is an important part of software development. Some functionality is data type specific, other functionality is defined on almost all data types and only depends on the structure of a data type. This is called *data type generic* functionality. Examples of data type generic functionality are: storing a value in a database, editing a value, comparing two values for equality, pretty-printing a value, etc. If a data type changes, or a new data type is added to a piece of software, a generic program automatically ‘adapts’ to the changed or new data type. Since a generic program automatically ‘adapts’ to a changed data type, a programmer only has to program the exception.

4 Generic programming

In Haskell it is very hard or even impossible to write a function that works on *all* data types. Haskell provides standard type classes offering functions that can be overloaded by writing an instance declaration. For example, the *Eq* class provides the equality operator that can be overloaded for newly defined data types. However, each instance declaration must contain a new implementation of the function being overloaded. It is possible to automatically derive some functions, like equality, parsers and pretty printing. But this mechanism is closed and cannot be extended by the programmer. Generic programming provides a way to define *generic functions* and allows the programmer to abstract over the *structure* of a type. It is possible to develop functions that work for a range of types, even on ones that have not yet come to pass. Generic programming makes it possible to solve a class of problems once and for all, instead of writing new code for every single instance. This fact increases re-usability and reliability, the latter due to the fact that a function has probably already been used and tested on several types. Generic programming challenges programmers to find ‘generic’ components, strip them of irrelevant detail and write a generic function.

The examples in this chapter, and in the remainder of this document, are based on the ‘Extensible and Modular Generics for the Masses (EMGM)’ library [6]. What characterizes this library is that it doesn’t need any extensions of Haskell (like the ones reviewed in Chapter 3), it can be completely defined in Haskell98. This feature makes the library excel in portability.

4.1 Type-indexed functions

Type-indexed functions are the first step toward generic functions. In order to define a type-indexed function it is necessary to create some sort of *type representation*. This is needed because it is not possible (or at least excruciatingly hard) to index a function directly on Haskell types. Type-indexed functions are indexed on the type representation of types.

There exist various forms of type representation. Some are based on GADTs, see Section 3.1, while others are type class based. EMGM has a type class based type representation; its *Generic* class contains all the represented types and serves as a blueprint for type-indexed and generic functions:

```
class Generic g where
  char :: g Char
  int  :: g Int
  list :: (GRep a)  $\Rightarrow$  g [a]
  prod :: (GRep a, GRep b)  $\Rightarrow$  g (Prod a b).
```

Furthermore a class *GRep* is used as a kind of dispatcher, relaying func-

4.1 Type-indexed functions

tion calls to the appropriate member function of the *Generic* class:

```
class GRep a where
  over :: (Generic g) ⇒ g a
instance GRep Int where
  over = int
instance GRep Char where
  over = char
instance (GRep a) ⇒ GRep [a] where
  over = list
instance (GRep a, GRep b) ⇒ GRep (Prod a b) where
  over = prod.
```

The introduced type classes can handle only functions that have *one* generic type variable (arity 1 functions, like a generic sum). Whereas the real EMGM library, used in the tests (Section 8), can handle functions requiring three generic type variables (arity 3 functions, like the generic *ZipWith* function). Theoretically it is even possible to go further up in arity. However, functions with that kind of arity are very rare.

With the type representation installed, we can proceed to construct type-indexed functions. A clear example of a type-indexed function is *sum*. This function adds all the integer values in a data structure. Before implementing *sum*, the signature of the type indexed function has to be given. Rather unusual, a **newtype** declaration specifies the type of the function:

```
newtype Sum a = Sum { sum' :: a → Int }.
```

It is important to read the above declaration as a type signature; the **newtype** declaration is just an idiom for embedding generics in Haskell98.

It is known that the type-indexed function *sum* itself cannot be a genuine polymorphic function of type $a \rightarrow Int$. The *sum* function does not work for arbitrary types, but only for types that are representable. Consider the following definition for *sum*:

```
sum :: (GRep a) ⇒ a → Int
sum = sum' over
```

```
instance Generic Sum where
  int   = Sum (\x → x)
  char = Sum (\x → 0)
  list = Sum (\x → case x of
                [] → 0
                x : xs → sum x + sum xs)
  prod = Sum (\x → sum (outl x) + sum (outr x)).
```

4 Generic programming

With *sum* it is possible to add all integers in, for example, a list of *prods*¹:

```
> sum [(Prod 'a' 1), (Prod 'b' 2), (Prod 'c' 3)]
6.
```

The *sum* function is a type-indexed function but not a *generic* function. The distinction between type-indexed functions and generic functions is that a type-indexed function works for all representable types (members of the type class *Generic*), whereas a true generic function works for *all* types. To make *sum*, or any other function, generic it is necessary to extend the type representation with a *view* type that is isomorphic to the Haskell data definition (e.g. the sum-of-products view [19]). With a ‘view’ type we’re only half way there, what remains is a way to convert an ordinary data type to the ‘view’ data type and vice versa. A set of functions from and to the ‘view’ data type are required. Such a set is called an *embedded projection pair*.

4.2 Generic algorithms

This section explores two algorithms, one for a sum and one for equality. The latter has been introduced in the previous section. These algorithms are generic in a sense that they operate on a large group of data types. However, it is still necessary to give an instance declaration for every data type.

Sum. To use the *sum* function on binary trees and rose trees, it is necessary that those types become representable. For convenience, their definitions are repeated here:

```
data BinTree a = Leaf a | Bin (BinTree a) (BinTree a)
data Rose a   = Node a [Rose a].
```

To make these data types representable, they have to be added to the type representation type class. Since type classes are not extensible, the ‘types’ alias member functions have to be declared at the same place as the *Generic* class. This is obviously a disadvantage that will also be alleviated with generic functions. The next section reveals how this is done. For now we make a new definition of the *Generic* type representation class, including the new data types:

```
class Generic g where
  char  :: g Char
```

¹where *Prod* is defined as: `data Prod = Prod { outl :: a, outr :: b }`, more about this data type in Section 4.3.

```

int    :: g Int
list   :: (GRep a) => g [a]
prod   :: (GRep a, GRep b) => g (Prod a b)
bintree :: (GRep a) => g (BinTree a)
rose   :: (GRep a) => g (Rose a)

```

the accompanying instance declarations for the dispatcher:

```

instance (GRep a) => GRep (BinTree a) where
  over = bintree
instance (GRep a) => GRep (Rose a) where
  over = rose

```

the redefinition of *sum* containing the new data types:

```

instance Generic Sum where
  int    = Sum (\x -> x)
  char   = Sum (\x -> 0)
  list   = Sum (\x -> case x of
                    [] -> 0
                    x : xs -> sum x + sum xs)
  prod   = Sum (\x -> sum (outl x) + sum (outr x))
  bintree = Sum (\x -> case x of
                    Leaf y -> sum y
                    Bin l r -> sum l + sum r)
  rose   = Sum (\(Node x rs) -> sum x + sum rs).

```

Now it is possible to apply the sum function on binary trees:

```

> sum (Bin (Bin (Leaf 32) (Leaf 3)) (Leaf 4))
39.

```

Equality. The holy grail of generic programming is arguably the *equality* function: it decides whether two given values of the same data type are equal to each other. We inspect the implementation of equality by means of a type-indexed function *eq*:

```

newtype Eq a = Eq { appEq :: a -> a -> Bool }

```

```

instance Generic Eq where
  int    = Eq (\x y -> x == y)
  char   = Eq (\x y -> x == y)
  list   = Eq (\x y -> case (x, y) of
                    (x : xs, y : ys) -> x 'eq' y & xs 'eq' ys

```

4 Generic programming

```

                                ([], [])      → True
                                -            → False)
prod   = Eq (λx y → (outl x) 'eq' (outl y) ∧ (outr x) 'eq' (outr y))
bintree = Eq (λx y → case (x, y) of
                        (Leaf l, Leaf r)      → l 'eq' r
                        (Bin ll lr, Bin rl rr) → ll 'eq' rl ∧ lr 'eq' rr
                        -                      → False)
rose   = Eq (λx y → case (x, y) of
                        (Node x xs, Node y ys) → x 'eq' y ∧ xs 'eq' ys
                        -                      → False)
```

```
eq :: (GRep a) ⇒ a → a → Bool
eq = appEq over.
```

An example application of the `eq` function:

```
> eq (Node 1 [Node 2 [], Node 3 [[]]] (Node 1 [Node 2 [], Node 4 [[]]])
False.
```

We can see a recurring pattern in the definition of the cases for the new data types: follow the structure of the (recursive) data type and apply the function on the (primitive) elements. Generic functions come to aid and make the tiresome, and therefore error-prone, implementation of boilerplate code unnecessary.

4.3 Generic functions

Implementing a function so that it works for arbitrary data types may seem very daunting. However, it suffices to define such a function for primitive types, such as `Char` and `Bool`, and for three elementary types: the one-element type, the binary sum and the binary product:

```
data Unit      = Unit
data Plus a b = Inl a | Inr b
data Prod a b = Prod { outl :: a, outr :: b }.
```

Haskell's construct for defining new types, the data declaration, introduces a type that is *isomorphic* to a *sum of products*. If it is known how to process sums and products, it is also known how to process elements of an arbitrary data type. More generally, if a type T can be handled, then a representation type R that is isomorphic to T can be handled.

What is an isomorphism? A data type in Haskell can be seen as an algebra. When two algebras are isomorphic they are 'essentially' the same. More precise, two algebras A and B are called isomorphic whenever there

exists a *bijection* between A and B . This means that A and B are isomorphic when there exist functions $f :: A \rightarrow B$ and $g :: B \rightarrow A$ that cancel each other, that is:

$$\begin{aligned} f \cdot g &= id_B \\ g \cdot f &= id_A \end{aligned}$$

In other words, the two algebras can be converted to one another. An isomorphic relation can be expressed in Haskell as follows:

```
data Iso a b = Iso {from :: b -> a, to :: a -> b}.
```

In what follows b will always be the original data type and a its representation type.

We must adapt the *Generic* class once again to incorporate the sum-of-product types and a ‘view’ type that redirects the non-primitive data types to the sum-of-products types:

```
class Generic g where
  unit :: g Unit
  plus :: (GRep a, GRep b) => g (Plus a b)
  prod :: (GRep a, GRep b) => g (Prod a b)
  view :: (GRep a) => Iso a b -> g b
  char :: g Char
  int  :: g Int.
```

The seven member functions correspond to the elementary (‘view’) types, *Unit*, *Plus*, *Prod* and to a small selection of primitive types, *Char* and *Int*. The member function *view* slightly breaks ranks and deals with arbitrary data types. Each method binding defines the instance of the generic function for the corresponding type.

The following instance declarations make the dispatcher complete:

```
instance GRep Unit where
  over = unit
instance (GRep a, GRep b) => GRep (Plus a b) where
  over = plus.
```

Notice there isn’t an instance declaration that uses ‘view’, this member function is merely used to convert the data type to its view (sum-of-products) isomorphic counterpart.

To be able to use a generic function on binary and rose trees, they have to be made representable. Here are the necessary instance declarations:

```
instance (GRep a) => GRep (BinTree a) where
  over = view (Iso fromBinTree toBinTree)
```

4 Generic programming

```

fromBinTree :: BinTree a → Plus a (Prod (BinTree a) (BinTree a))
fromBinTree (Leaf x) = Inl x
fromBinTree (Bin l r) = Inr (Prod l r)

```

```

toBinTree :: Plus a (Prod (BinTree a) (BinTree a)) → BinTree a
toBinTree (Inl x) = Leaf x
toBinTree (Inr (Prod l r)) = Bin l r

```

```

instance (GRep a) ⇒ GRep (Rose a) where
  over = view (Iso fromRose toRose)

```

```

fromRose :: Rose a → Prod a [Rose a]
fromRose (Node x rs) = Prod x rs

```

```

toRose :: Prod a [Rose a] → Rose a
toRose (Prod x rs) = Node x rs.

```

The remainder of this paragraph implements the two functions, *sum* and *eq*, as true generic functions, *gsum* and *geq* respectively.

Generic equality. An element of *Geq a* is an instance of *Geq* that compares values of type *a* for equality. The function is defined as follows:

```

newtype Geq a = Geq { appGeq :: a → a → Bool }

```

```

geq :: (GRep a) ⇒ a → a → Bool
geq = appGeq over.

```

In a way the generic function is applied to the type representation *GRep*. Of course, this is not the whole story. The code above defines only a convenient shortcut. The actual definition of *geq* is provided by an instance declaration, but it can be read as just a generic definition:

```

instance Generic Geq where
  unit    = Geq (λx y → case (x, y) of
                    (Unit, Unit) → True)
  plus    = Geq (λx y → case (x, y) of
                    (Inl xl, Inl yl) → geq xl yl
                    (Inr xr, Inr yr) → geq xr yr
                    _                → False)
  prod    = Geq (λx y → geq (outl x) (outl y) ∧
                    geq (outr x) (outr y))

```

4.3 Generic functions

```
view iso = Geq (λx y → geq (from iso x) (from iso y))
char     = Geq (λx y → x ≡ y)
int      = Geq (λx y → x ≡ y).
```

The only case that can occur when comparing elements of the type *Unit* is to compare *Unit* to *Unit* which yields *True*. To compare elements of a sum type, it is only necessary to compare values of the same constructor (either *Inl* or *Inr*, as defined in the data definition of *Plus*) otherwise it yields *False*. The comparison of a pair is given by the conjunction of the pairwise comparison of the components. The primitive types make use of the equality function from the *Eq* class.

Generic sum. The implementation for the generic sum is constructed in a similar way:

```
newtype Gsum a = Gsum { appGsum :: a → Int }
```

```
gsum :: (GRep a) ⇒ a → Int
gsum = appGsum over
```

instance Generic Gsum where

```
unit     = Gsum (λx → 0)
plus     = Gsum (λx → gsum (Inr x) + gsum (Inl x))
prod     = Gsum (λx → gsum (outl x) + gsum (outr x))
view iso = Gsum (λx → gsum (from iso x))
char     = Gsum (λx → 0)
int      = Gsum (λx → x).
```

Notice that the implementation of *gsum* in the evaluation test for EMGM (discussed in Chapter 8), is defined in a even more generic way. A very powerful generic function *reduce* is used to express the generic *sum* function.

Part II
Criteria

Chapter 5

Research Goal

Defining functions and data types is an important part of software development. Some functions are defined on a limited set of data types, like *IsZero* for data types in the *Num* class. Other functions work for many or even all data types, like the *show* function that prints a data type on a screen. A function that works for many data types and on the *structure* Of a data type is called a *generic function* as explained in detail in Chapter 4.

5.1 Problem description

Generic programming is important for the following reasons: generic programming removes the need of implementing *boilerplate* code over and over again. In contrast to overloading, generic programming allows the programmer to construct a *single* function that works on many types. Generic functions are built to last, they even work for data types that haven't been defined yet.

The requirements for a software system are likely to be extended or changed during the lifetime of a software system. Changing an existing software system is often very difficult; most software is designed in such a way that entities in a software system depend on each other. Keeping a software system, including parts like documentation and interfaces, in a consistent and stable state is not an easy task. Software developers should take into account that software will evolve during its lifetime [35]. A generic program, however, is able to handle a new or changed data type automatically. Generic programming has the potential to solve at least an important part of the software-evolution problem [23].

Data type generic programming was introduced more than ten years ago. Generic programming has made a lot of progress in that period, especially in the context of Haskell (Section 2.3). Since 2000 at least ten generic programming libraries have been introduced.

Although generic programming is used in several applications, large soft-

5 Research Goal

ware projects hesitate to use it in real life¹. This is probably due to the fact that developing a large software system takes a long time and choosing a particular approach to generic programming for such a project involves a risk. Few approaches that have been developed over the last decade are still supported and there is a high risk that the chosen approach will not be supported anymore, or that it will change in a backwards-incompatible way in a couple of years time.

Furthermore, it is often not immediately clear which approach is best suited for a particular project. There are generic functions that are difficult or impossible to define in certain approaches. The set of data types to which a generic function can be applied varies among different approaches and the amount of work a programmer has to do per data type and/or generic function varies as well.

The current status of generic programming in Haskell is comparable to the birth of Haskell in the eighties [21]. We have many libraries, each individually lacking critical mass in terms of language/library-design effort, implementations and users.

5.2 Goal

The risk in using generic programming can be reduced by designing a *common generic programming library* for which continuing support is guaranteed. Continuing support and consensus about the library can only be achieved by developing this library in an international committee. Such an international committee has already been assembled and the members stated that they are willing to participate; however, the collaboration is still in its ‘early stages’. The reason for developing a library instead of a language extension is that Haskell is powerful enough to support most generic programming concepts by means of a library. Compared with a language extension, a library is much easier to ship, support and maintain. The library might be accompanied by tools that depend on non-standard language extensions, for example for generating embedding-projection pairs, as long as the core is standard Haskell. The library should support the most common generic programming scenarios, so that programmers can define the generic functions that they want and use them with the data types they want.

The development of the library should be a community process. A community is essential for using, maintaining, documenting and, very important, advertising the common library. To achieve this ultimate goal it is necessary to combine the effort of the different parties involved. It is mandatory to get a large part of the key players in the generic programming scene along side.

¹There are a few exceptions, for example the Haskell Refactorer [36] uses Strafinski [34].

The development of the common library and community related actions are beyond the scope of this research. This thesis' main focus is on the first step towards a common generic programming library: the definition of a set of relevant criteria and an evaluation procedure (test suite) to asses them. The evaluation and comparison of generic programming libraries is a secondary goal.

Research questions. In order to reach the given goal, we have to answer the following questions:

1. What are the analysis criteria?
2. How do the different libraries score on these criteria?

5.3 Research approach

The first step towards a common generic programming library is to evaluate existing libraries to find out differences and commonalities, and to be able to make well-motivated decisions about including and excluding features. This thesis is about defining a set of criteria, and evaluating and comparing existing libraries for generic programming in Haskell. We will evaluate and compare the following libraries:

- Derivable type classes (DTCs)² [17],
- Extensible and Modular Generics for the Masses (EMGM) [6], based on Generics for the Masses (GM) [11] and Generics as a Library (GL) [12],
- Generic Programming, Now (NOW) [15],
- Light-weight implementation of Generics and Dynamics (LIGD) [4],
- Light-weight PolyP [44],
- RepLib [48],
- Scrap Your Boilerplate (SYB)³ [29, 30, 31],
- Smash Your Boilerplate (Smash) [28],
- SYB Revolutions [14], preceded by SYB Reloaded [16],
- Uniplate [42].

²DTCs is actually a language extension, but since it shares many characteristics with other libraries we will take this approach into consideration.

³For this paper we will only consider the latest version: SYB with class [31]

Scope. The list of generic programming approaches does not contain generic programming language extensions such as PolyP [22] or Generic Haskell [9, 39, 38] and no pre-processing approaches to generic programming such as DrIFT [49], Template Haskell [41] and *Data.Derive*. We strictly limit ourselves to library approaches, which, however, might be based on particular compiler extensions. The SYB [31] and Strafunski [33] approaches are very similar and therefore we only take the SYB approach into account in this evaluation. SYB reloaded [16] is another library that we omit, this library has been followed up by SYB revolutions.

Two other libraries that are redirected to future work, see Chapter 10 are: ‘A pattern for almost Compositional Functions (Compos)’ [3] (closely related to Uniplate) and ‘Yet Another Generics Scheme (YAGS)’ [7].

Hinze et al. [13] compare various approaches to generic programming in Haskell. However, most of the covered approaches are language extensions and many of the very recent library extensions have not been included. Our comparison does not take language extensions into account, unless they can easily be expressed as a library. When comparing only library approaches, we can formulate more precise criteria for comparison.

Evaluation. We evaluate existing libraries by means of a set of criteria. Criteria for generic programming can be extracted from papers about generic programming. Examples of such criteria are: can a generic function be extended with special behaviour on a particular data type and are generic functions *first-class*, for example, can they take a generic function as argument. We develop a set of criteria based on our own ideas about generic programming and ideas from papers about generic programming. For most criteria, we have a generic function (Chapter 8) that determines whether or not the criterion is satisfied. We have collected a set of generic functions for testing the criteria. We try to implement all of these functions in the different approaches.

A comparison of libraries is necessary for developing a common generic programming library, but is also interesting in itself. This thesis offers an in-depth evaluation of quite distinct generic programming approaches. These approaches range from *intensional* (based on *run-time* information) to *extensional* (based only on the type information at *compile-time*, specialisation happens at compile time); from approaches operating on representations of terms to approaches operating on the terms themselves.

Deliverables. The research will produce the following deliverables:

- an extensive set of well documented *criteria* for comparing libraries for generic programming in Haskell,
- a *generic programming test suite*: a set of characteristic test functions

5.3 Research approach

that you can use to test the library against the criteria for generic programming libraries,

- a preliminary evaluation and comparison of many of the existing library approaches to generic programming in Haskell with respect to the criteria, using the implementation of the test suite in the different libraries.

The result of this research is likely to be the most important input to the design process of the common generic programming library.

Chapter 6

Generic programming libraries

In this chapter we describe and examine the different libraries for generic programming in Haskell. In the following sections we briefly introduce each library and implement one test function as a showcase in every library. In this way we get a nice overview of the different forms of generic programming in Haskell.

The comparison of generic programming approaches by Hinze et al. [13] mentions that generic programming approaches can be sorted into the following groups:

- language extensions based on Hinze’s theory of *type-indexed functions* with *kind-indexed* types [9], like Generic Haskell [38] and Clean [45],
- approaches based on a kind of *reflection mechanism*, like DrIFT [49] and Template Haskell [41],
- light-weight approaches, like EMGM [6] and LIGD [4],
- other approaches, like SYB [31] and PolyP [22].

In contrast to their comparison, ours only includes libraries and omits the remaining approaches, like language extensions. We have altered the sorting, a bit, to our convenience. The presented libraries are divided into two groups with similar characteristics:

1. Light-weight approaches to generic programming:
 - Light-weight implementation of Generics and Dynamics,
 - Generic Programming, Now!,
 - Extensible and Modular Generics for the Masses,
 - RepLib,

6 Generic programming libraries

- Derivable Type Classes¹,
 - Light-weight PolyP,
2. Strategic programming based approaches:
- Scrap Your Boilerplate with class,
 - Scrap Your Boilerplate revolutions,
 - Smash Your Boilerplate,
 - Uniplate.

We examine the approaches in the groups together, since the libraries have many commonalities.

Many example usages exist for generic programming, for instance Generic Equality. We call those example usages *scenarios*. A scenario describes a particular kind of generic programming usage. Ideally a generic programming library should support many scenarios. A scenario is made explicit by means of a (or more) test function(s). For example the scenario Generic Read entails test functions that *produce* a value, like the *fulltree* function that produces a value of a container data type. This particular function is used in the efficiency test. The scenarios and related test functions are described in detail in Chapter 8.

In our showcase we use the FoldTree scenario that checks whether a library is able to lists all occurrences of elements of a particular constant type in a container data type. This scenario tests if a library is able to handle an *ad-hoc type case*, which specifies different behaviour for a specific data type.

We use the *foldSal* test function, from the FoldTree² scenario. The test function *foldSal* lists all *Salary* values that appear in a value of the *Company* data type. This relatively simple function can be defined in all approaches, with the exception of light-weight PolyP.

Recall the data types from Section 2.4; an illustrative company:

```
ouCom :: Company
ouCom = C [D "Research" jeuring [PU heeren, PU rodriguez],
          D "Board of Directors" gerdes []]
```

```
jeuring, rodriguez, heeren, gerdes :: Employee
jeuring  = E (P "Jeuring"   "Utrecht") (S 10000.0)
rodriguez = E (P "Rodriguez" "Utrecht") (S 2000.0)
heeren   = E (P "Heeren"   "Heerlen") (S 8000.0)
gerdes   = E (P "Gerdes"   "Emmen")   (S 500.0).
```

¹DTCs is actually a language extension, but since it shares many characteristics with the light-weight approaches, it falls into this group.

²A slightly better name is *Listify*, however for historical reasons we will stick to *FoldTree*

The result of *foldSal ouCom* should be:

```
[S 10000.0, S 8000.0, S 2000.0, S 500.0].
```

The definition of the *foldSal* test function in each library follows the next naming scheme: *foldSal < Library Abbreviation >*; ‘help’ functions can be distinguished from the actual test function by means of an apostrophe appended to the function name. We start with the examination of the light-weight approaches.

6.1 Light-weight approaches

Due to Haskell’s advanced type language and type classes it is possible to write generic programs in Haskell itself, without extending the language. An approach in which generic programs are plain Haskell programs is called a light-weight approach. Light-weight approaches to generic programming in Haskell have become popular in the last couple of years.

6.1.1 Derivable Type Classes

Derivable Type Classes (DTCs) [17] try to provide a richer language for *default method definitions* in a class declaration. This is supposed to give an elegant way to extend Haskell with the power of generic programming. The default methods are *generalised*.

A type class declaration corresponds roughly to the type signature of a generic definition; or rather, to a collection of type signatures. Instance declarations are related to the type cases of a generic definition. The crucial difference is that a generic definition works for all types, whereas instance declarations must be provided explicitly by the programmer for each newly defined data type. There is, however, one exception to this rule. For a handful of built-in classes Haskell provides special support, the so-called ‘deriving’ mechanism. For instance, if you define

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

then Haskell generates the ‘obvious’ code for equality. DTCs allows you to specify the ‘obvious’ precisely. Derivable type classes generalise this feature to arbitrary user-defined classes: generic definitions are used to specify *default methods* so that the programmer can define her own derivable classes.

A generic default method is defined on *type arguments*, enclosed in peculiar curly braces and work by *induction* over the *structure* of a type. It is necessary to give an instance declaration for every data type that is going to be applied. This might be considered a disadvantage but then again it can be used to control which data types are allowed (the universe).

Here is how the *foldSal* test function is implemented in DTCs:

6 Generic programming libraries

```
class FoldSalDTCs a where
  foldSalDTCs :: a → [Salary]
  foldSalDTCs { | Unit | } Unit = []
  foldSalDTCs { | a + b | } (Inl x) = foldSalDTCs x
  foldSalDTCs { | a + b | } (Inr y) = foldSalDTCs y
  foldSalDTCs { | a × b | } (x × y) = foldSalDTCs x ++
                                         foldSalDTCs y

instance FoldSalDTCs Int where
  foldSalDTCs x = []
instance FoldSalDTCs Char where
  foldSalDTCs x = []
instance FoldSalDTCs Float where
  foldSalDTCs x = [].
```

DTCs use the same sum-of-products view as EMGM, see Section 4.3. Generic functions are defined on three *elementary* types: the unit type, the binary sum and the binary product. The code above traverses the structure of a data type and concatenates the results. If no specialisation is provided the function returns an empty list.

To list all the *Salary* values, the function is specialised for that data type:

```
instance FoldSalDTCs Salary where
  foldSalDTCs x = [x].
```

The rest of the necessary instance declarations:

```
instance FoldSalDTCs a ⇒ FoldSalDTCs [a]
instance FoldSalDTCs Company
instance FoldSalDTCs Dept
instance FoldSalDTCs Unit
instance FoldSalDTCs Employee
instance FoldSalDTCs Person.
```

6.1.2 Extensible and Modular Generics for the Masses

The Extensible and Modular Generics for the Masses [6] (EMGM) is based on Generics for the Masses (GM) [11] approach. The library, explained in Section 4.3, is entirely defined in Haskell98 and does not need any language extensions. However, the library has the option to use a *generic dispatcher* (which is used to call the appropriate function from the *Generic* class, see Section 4.3), when multi-parameter type classes (Section 3.5) are allowed.

The implementation of the *foldSal* test function for EMGM:

```
newtype FoldSalEMGM a b c =
  FoldSalEMGM {foldSal' :: a → [Salary]}
```

The definition of the *foldSal* test function is based on the EMGM library with three generic type variables, in contrast to the one generic type variable library used in Chapter 4. The *foldSalEMGM* test function is of arity one, the remaining two generic type variables, *b* and *c* are discarded.

```
instance Generic FoldSalEMGM where
  unit          = FoldSalEMGM (λx → [])
  plus a b     = FoldSalEMGM (λx → case x of
                                Inl l → foldSal' a l
                                Inr r → foldSal' b r)
  prod a b     = FoldSalEMGM (λx → foldSal' a (outl x) ++
                                foldSal' b (outr x))
  view iso _ _ a = FoldSalEMGM (λx → foldSal' a (from iso x))
  int          = FoldSalEMGM (λx → [])
  char        = FoldSalEMGM (λx → [])
  float       = FoldSalEMGM (λx → [])
instance GenericCompany FoldSalEMGM where
  salary      = FoldSalEMGM (λx → [x])
```

The class *GenericCompany* is a subclass of *Generic* and is used to provide an ad-hoc instance for the *Salary* data type. The definition of the test function *foldSalEMGM*:

```
foldSalEMGM :: GRep FoldSalEMGM a ⇒ a → [Salary]
foldSalEMGM = foldSal' over.
```

6.1.3 Generic Programming, Now!

Generic Programming, Now!(NOW) [15] is a relatively new (2007) library that uses some recent developments in programming language research: GADTs, open data types and open functions [40]. An *open data type* and an *open function* need not be defined at a single location, but the definitions may be scattered around the program. The order of function equations is determined by best-fit pattern matching, where a specific pattern takes precedence over an unspecific one.

NOW uses an open GADT for type representations. Using this data type, a programmer can add a new type representation by adding a constructor to the open GADT. Since the type representation data type is open, a new type representation can be added anywhere in a program and the library doesn't have to be updated for every new data type that is made representable. Open functions are used to extend a function to cover new

6 Generic programming libraries

constructors (representation types). A fundamental problem with this approach is that open data types and open functions are not supported in Haskell or one of its extensions. As long as there is no support for open data types and functions, each new type representation requires a change of the library. Here is an excerpt of the (open) type representation GADT:

```
data Type :: * -> * where
  CharR    :: Type Char
  IntR     :: Type Int
  PairR    :: Type a -> Type b -> Type (a, b)
  ListR    :: Type a -> Type [a]
  SpineR   :: Type a -> Type (Spine a)
  (↦)      :: Type a -> Type b -> Type (a -> b)
  CompanyR :: Type Company
  DeptR    :: Type Dept
  UnitR    :: Type Unit
  EmployeeR :: Type Employee
  PersonR  :: Type Person
  SalaryR  :: Type Salary.
```

The data type *Typed* combines a value with its representation type:

```
infixl 1 :>
data Typed a = (:>) { typeOf :: Type a, val :: a }.
```

The NOW library supports multiple views on data types: (variants of) the spine view, but also the sum-of-products view. The spine view is the default view. For each data type a translation to the spine view has to be provided. Here is the definition of the *Spine* structure type :

```
data Spine :: * -> * where
  Con :: Constr a -> Spine a
  (◇) :: Spine (a -> b) -> Typed a -> Spine b
```

and the type representation for the list data type:

```
toSpine :: Typed a -> Spine a
toSpine (ListR a :> []) = Con nil
toSpine (ListR a :> x : xs) = Con cons ◇ (a :> x) ◇ (ListR a :> xs)
```

```
nil = Descr { constr = [],
              name = "[]",
              arity = 0,
              fixity = Prefix 10,
              order = (0, 2) }
```

```

cons = Descr { constr = (:),
              name   = "(:)",
              arity  = 2,
              fixity  = Prefix 10,
              order  = (1, 2)}.

```

The inverse function from the structure type back to the original data type is a parametric polymorphic function that works for all represented data types.

With the necessary machinery installed, we can implement the *foldSal* test function:

```

foldSalNOW' :: Typed a → [Salary]
foldSalNOW' (SalaryR :> s)      = [s]
foldSalNOW' (SpineR a :> Con c) = []
foldSalNOW' (SpineR a :> (f ◇ x)) =
  foldSalNOW' (SpineR (typeOf x ↦ a) :> f) ++ foldSalNOW' x
foldSalNOW' x                    =
  foldSalNOW' (SpineR (typeOf x) :> toSpine x)

```

```

foldSalNOW c = foldSalNOW' (CompanyR :> c).

```

The cases indexed with *SpineR* traverse the data structure and return the empty list in case of a non-recursive type. This behaviour is specialised with the case indexed on *SalaryR*, which returns a singleton list containing the *Salary* value.

6.1.4 LIGD

Light-weight Implementation of Generics and Dynamics (LIGD) [4] is an approach to embedding generic functions and dynamic values into Haskell. It only needs Haskell extended with existential types. The dynamics part of LIGD is not relevant for the discussion of generics and is not discussed here. The interested reader is referred to the original paper [4].

The LIGD library is similar to EMGM, it reflects the type argument onto the value level so that the type case can be implemented by ordinary pattern matching. LIGD uses a parametric type for type representations: *Rep t* → *t*. Here *Rep t* is the type representation of *t*. The type representation *Rep* uses equivalence types:

```

data Rep t =
  | RInt   (EP t Int)
  | RChar (EP t Char)
  | RFloat (EPT tT Float)
  | RUnit  (EP t Unit)
  | ∀ a b. RSum (Rep a) (Rep b) (EP t (Sum a b))

```

6 Generic programming libraries

```

|  $\forall a b. RPair (Rep a) (Rep b) (EP t (Pair a b))$ 
|  $\forall a. RType Term (Rep a) (EP t a)$ 
|  $RCon String (Rep t)$ .

```

The constructors *RInt* and *RChar* represent the primitive types and the constructors *RUnit*, *RSum* and *RPair* the structure types. The constructor *RType* is used for representing user-defined data types and *RCon* for constructor information.

An example of a representable type, a type on which a generic function can be used:

```

list :: Rep a → Rep (List a)
list a = Type ((Con "Nil" unit) + (Con "Cons" (a * (list a))))
          (EP fromList toList)

```

where *unit*, *+* and *** are smart versions of the respective constructors (defined in the LIGD library) and *fromList* and *toList* convert between the type *List* and its structure type:

```

fromList :: List a → Unit + (a × (List a))
fromList Nil           = Inl Unit
fromList (Cons a as) = Inr (a × as)

```

```

toList :: Unit + (a × (List a)) → List a
toList (Inl Unit)   = Nil
toList (Inr (a × as)) = Cons a as.

```

Note that the representation of the structure type records the name of the constructors.

The next piece of code gives the definition of the *foldSal* test function:

```

foldSalLIGD' :: Rep a → a → [Salary]
foldSalLIGD' (RInt ep)          i = []
foldSalLIGD' (RSum rA rB ep) t = case from ep t of
    Inl a → foldSalLIGD' rA a
    Inr b → foldSalLIGD' rB b
foldSalLIGD' (RPair rA rB ep) t = case from ep t of
    (a × b) → foldSalLIGD' rA a
    + foldSalLIGD' rB b
foldSalLIGD' (RType e rA ep) t = foldSalLIGD' rA (from ep t).

```

LIGD does not allow for a nice specification of ad-hoc cases, so we are forced to make the next hack:

```

foldSalLIGD' (RCon "S" a) t = case a of
    RFloat ep → [S (from ep t)]

```

$$\begin{aligned} \text{foldSalLIGD}' (RCon\ s\ rA)\ t &= \text{foldSalLIGD}'\ rA\ t \\ \text{foldSalLIGD}'\ _ & \quad t = [] \end{aligned}$$

$$\begin{aligned} \text{foldSalLIGD} &:: \text{Company} \rightarrow [\text{Salary}] \\ \text{foldSalLIGD} &= \text{foldSalLIGD}'\ r\text{Company}. \end{aligned}$$

6.1.5 Light-weight PolyP

The pre-processor-based language extension PolyP [22] was later packaged up as a light-weight library [44].

The library provides generic functionality for regular data types of kind $\star \rightarrow \star$ (with one parameter). A data type is regular if it does not contain *function spaces* and if the arguments of the data type constructor on the left- and right-hand sides in its definition are the same. Examples of regular data types are *List a*, *Rose a* and *Fork a*. The data types *CharList*, *Tree* and *GRose* are regular, but have kind \star , \star and $(\star \rightarrow \star) \rightarrow \star \rightarrow \star$, respectively. The library cannot handle mutually recursive data types, so the set of data types (the universe) supported is relatively small. This means that the library does not pass the *foldSal* test, since the *Company* data type is of kind \star .

However, a smaller set of data types makes it possible to express more generic functions; the library contains definitions of *fold* and *unfold*, traversals and even functions generic in two type parameters like *transpose* :: $\dots \Rightarrow d\ (e\ a) \rightarrow e\ (d\ a)$.

The limited set of data types means that light-weight PolyP is not suitable as a general generic library.

The light-weight PolyP approach uses a combination of the fixed-point view and the sum-of-products view.

6.1.6 RepLib

Some type class instances can be automatically derived from the structure of types. As a result, the Haskell language includes the *derive* mechanism to automatically generate such instances for a small number of built-in type classes. RepLib [48] enables a similar mechanism for arbitrary type classes. RepLib defines the relationship between the structure of a data type and the associated instance declaration, of those arbitrary classes, by a normal Haskell function that pattern matches a representation type. Operations defined in this manner are extensible; instances for specific types not defined by type structure may also be incorporated.

The deriving-like behaviour works by using Template Haskell (TH) to define representation types that programmers may use to specify the default behaviour of type-indexed operations. Representation types reflect the

6 Generic programming libraries

structure of types as Haskell data, therefore programmers can define type-indexed operations as ordinary Haskell functions. The representation type of the *Company* data types can be derived as follows:

```
$ (derive
  [ ' ' Company,
    ' ' Dept,
    ' ' Unit,
    ' ' Employee,
    ' ' Person,
    ' ' Salary])
```

```
rName :: R Name
rName = rep
rAddress :: R Address
rAddress = rep
rManager :: R Manager
rManager = rep.
```

Notice that the types *Name*, *Address* and *Manager* cannot be derived. The *derive* function isn't able to handle **type** declarations. The solution to this deficiency is fairly simple, just use the dispatcher. The dispatch function *rep* dispatches the function call to the correct class method.

RepLib uses a GADT to represent types and a type class to support convenient access to them. The generic view of data types consists of a constructor data type, containing an embedded projection pair and a representation of the type list and a data type (*DT*) containing information about the data type itself (like the name).

The following code implements the *foldSal* test function:

```
data FoldSalRepLib a =
  FoldSalRepLib { foldSalRepLib' D :: a → [Salary] }

class Rep1 FoldSalRepLib a ⇒ FoldSalRepLib' a where
  foldSalRepLib' :: a → [Int]
  foldSalRepLib' = foldSalRepLib' R1 rep1

foldSalRepLib' R1 :: R1 FoldSalRepLib a → a → [Salary]
foldSalRepLib' R1 rSalary1      x = [x]
foldSalRepLib' R1 (Data1 dt cons) x = case findCon cons x of
  Val emb rec kids →
    foldl_1 (λca a b → a ++ (foldSalRepLib' D ca b)) [] rec kids
foldSalRepLib' R1 _      x = []
```

Some of the type names are appended, indicating that it can only handle functions of arity one. The *foldl1* function is the counterpart of *foldl* on type lists.

instance *FoldSalRepLib'* *a* \Rightarrow *Sat* (*FoldSalRepLib* *a*) **where**
dict = *FoldSalRepLib* *foldSalRepLib'*.

RepLib makes use of explicit dictionaries by means of the type class *Sat* for parameterised representation. The interested reader can find the details in the paper [48].

The case indexed on the view type *Data1* is used to traverse the data type structure. The behaviour is specialised for the *Salary* data type, resulting in a singleton list. The last case is a ‘catch-all’ where the remaining data types are matched.

Just like DTCs, instance declarations have to be given for applicable data types:

instance *FoldSalRepLib'* *Float*
instance *FoldSalRepLib'* *Int*
instance *FoldSalRepLib'* *Char*
instance *FoldSalRepLib'* ()
instance (*FoldSalRepLib'* *a*, *FoldSalRepLib'* *b*) \Rightarrow *FoldSalRepLib'* (*a*, *b*)
instance (*FoldSalRepLib'* *a*) \Rightarrow *FoldSalRepLib'* [*a*]
instance *FoldSalRepLib'* *Company*
instance *FoldSalRepLib'* *Dept*
instance *FoldSalRepLib'* *Unit*
instance *FoldSalRepLib'* *Employee*
instance *FoldSalRepLib'* *Person*
instance *FoldSalRepLib'* *Salary*

foldSalRepLib :: *FoldSalRepLib'* *a* \Rightarrow *a* \rightarrow [*Salary*]
foldSalRepLib = *foldSalRepLib'*.

The RepLib library also offers SYB-like functions like *everything* and *mkQ*. These functions can be used to implement a much less verbose alternative:

foldSalRepLib'' :: *Company* \rightarrow [*Salary*]
foldSalRepLib'' = *everything* (++) ([] ‘mkQ’ (:[])).

6.2 Strategic Programming

Strategic programming [32] in our context is generic programming with the use of strategies. A strategy, in the context of generic programming, is a

generic data processing action that can traverse into heterogeneous data structures while mixing uniform and type specific behaviour. Computing with heterogeneous data structures involves traversing them and performing actions at every node in the structure. Some actions may involve combining results of other actions into a value, or transforming a value (so-called *type preserving* computations such as *maps*). Using a combinator style, traversal schemes can be defined, and actual traversals are obtained by passing the problem specific input as parameters to suitable schemes.

Strategic programming offers a solution based on the following key ideas: action at a node and traversal must be separated and traversals must be decomposed into one-layer traversals on the one side and recursion schemes given by an explicit fixed-point definition on the other side. This separation of concerns permits composition and parametrisation: actions can be composed and one-layer traversals and recursion schemes can be provided as action-parametric generic combinators.

In order for an action at a node to be generic, strategic programming assumes the possibility of type-based dispatch, usually implemented in terms of a dynamic type-case, i.e. code that enquires about a value's type at run-time in order to perform an appropriate computation.

In contrast to the light-weight approaches, discussed in the previous section, most strategic programming approaches make use of the *Data.Generics* library provided by GHC.

6.2.1 Scrap Your Boilerplate

There are three versions of Scrap Your Boilerplate (SYB) [31, 29, 30]. The first SYB version presents a *type-safe cast* operator, it is implemented using a cast operator that cannot cause run-time errors. The implementation of *cast* relies on clever type class and reflection tricks that enable it to determine the type of a value at run-time by means of applying the instance for that type of an overloaded function *typeOf* that has been figured out at compile time by the type checker. More precisely, the cast operator has type: $cast :: (Typeable\ a, Typeable\ b) \Rightarrow a \rightarrow Maybe\ b$ where the *Typeable* type class declares the aforementioned *typeOf* function. The definition of *typeOf* is derived automatically by the compiler for a data type using Haskell's deriving clause.

Operationally, the application *cast x* within a context of the *Maybe T* type, returns *Just x* if *x* has type *T*, otherwise returns *Nothing*:

```
(cast 1) :: Maybe Int
> Just 1
(cast 1) :: Maybe Char
> Nothing.
```

In other words, we get a *Just* value when, at run-time, $a = b$.

SYB introduces several operators that can be seen as type-based dispatches, implemented in terms of *cast*: *mkT* (or ‘make transformation’) for type-preserving actions and *mkQ* for type-unifying actions. For example, given a function *f* of type $a \rightarrow a$, *mkT f x* applies *f* to *x* only if *x* has type *a*, returning *x* otherwise. Function *mkT* lifts a transformation on a value of a fixed type into a transformation on a value of type $\forall a. \text{Typeable } a \Rightarrow a$. It is therefore called a generic transformation.

The last version of SYB [31], which we will consider for the evaluation, shows that using type classes rather than run-time type casts can make generic programming using SYB more flexible. Each generic function is then defined as a class with default behaviour and type-specific behaviour can be added by defining specific instances of the defined class. This approach is a bit more verbose than the previous version, but has a significant advantage: instances of classes can be added in a modular way, also at a later stage. There is the possibility to extend an already existing generic function with new behaviour, without modification of existing code.

The definition of the *foldSal* test function:

```
foldSalSYB :: Company → [Salary]
foldSalSYB = everything (+) ([ 'mkQ' (:[])])
```

certainly looks familiar. We have seen this solution before in the implementation of the test using the RepLib library, in Section 6.1.6. An alternative definition using the *listify* function from the *Data.Generics* library:

```
foldSalSYB' :: Company → [Salary]
foldSalSYB' = listify (const True).
```

6.2.2 Smash Your Boilerplate

The Smash Your Boilerplate (Smash) [28] approach is conceptually closely related to Scrap Your Boilerplate approach. The latter uses a ‘type case’ operation based on the run-time type representation (*Typeable*). The Smash Your Boilerplate approach uses a *compile-time* type case operation. In both approaches, a new data type is presented to the library by declaring an instance of a special class: *Data* in SYB, *LData* in Smash.

In Smash a generic function is (quite literally) made of two parts. First, there is a term traversal strategy. One strategy may be to ‘reduce’ a term using a supplied reducing function (e.g. fold over a tree, like our *foldSal* test). Another strategy may rebuild a term. The second component of a generic function is *spec*, the list of ‘exceptions’, or ad-hoc redefinitions. Each component of *spec* is a function that tells how to transform a term of a specific type. Exceptions override the generic traversal.

The next piece of code shows the implementation of the *foldSal* test function using the Smash Your Boilerplate library:

6 Generic programming libraries

```
foldSalSmash xs =
  gmapq (SCons (λ(s :: Salary) → [s]) SNil) (concat) xs.
```

The $(\lambda(s :: \text{Salary}) \rightarrow [s])$ is the ad-hoc redefinition, and defines the behaviour for the *Salary* type.

6.2.3 SYB Revolutions

In their SYB Reloaded and Revolutions [14] papers, Hinze, Löh and Oliveira [42, 41] demonstrate that SYB’s *gfoldl* function is in essence a catamorphism on the *Spine* data type, which can be defined as follows:

```
data Spine a where
  Constr :: Constr → a → Spine a
  (◇)    :: Data a ⇒ Spine (a → b) → a → Spine b.
```

The spine view treats data uniformly as constructor applications; it is, in a sense, value-oriented. This is in contrast to the classical views: *fixed-point* (light-weight PolyP) and *sum-of-products* (EMGM), which can be characterised as *type oriented*. One distinct advantage of the spine view is its generality: it is applicable to a large class of data types, including generalised algebraic data types (Section 3.1). The reason for the wide applicability is: a data type describes how to construct data, the spine view captures just this. Its main weakness also roots in the value orientation: one can only define generic functions that *consume* data (like the *show* function) but not ones that *produce* data (read). The reason for the limitation is: a uniform view on individual constructor applications is useful if you have data in your hands, but it is of no use if you want to construct data.

SYB Revolutions provides a ‘type spine’ type, which makes it possible to define functions that produce data. Furthermore, a ‘lifted spine’ type is given for generic functions that are parametrised over type constructors. For example, using the lifted spine type, *map* and *reduce* can be defined.

Using the SYB Revolutions library the *foldSal* test function can be defined in the following manner:

```
type Query r = ∀a.Type a → a → r

mapQ :: Query r → Query [r]
mapQ q t x = mapQ' q $ toSpine (t :> x)

mapQ' :: Query r → (∀a.Spine a → [r])
mapQ' q (Con c)      = []
mapQ' q (f ◇ (t :> x)) = mapQ' q f ++ [q t x]
```

```

everything :: (r → r → r) → Query r → Query r
everything op q t x =
  foldl1 op ([q t x] ++ mapQ (everything op q) t x).

```

The above code implements a central SYB combinator *everything* that is used to construct generic queries. Using this combinator we can define *foldSalSpine* in a similar manner to SYB:

```

foldSalSpine :: Company → [Salary]
foldSalSpine = everything (++) ([ 'mkQ' (:[]) ] CompanyR
  where
    mkQ :: [Salary] → (Salary → [Salary]) → Type a → a → [Salary]
    mkQ zero lift SalaryR i = lift i
    mkQ zero lift _ _ = zero.

```

The *CompanyR* is the type representation of the *Company* data type.

6.2.4 Uniplate

Uniplate uses a *Uniplate* class that abstracts over common traversals and queries. The *Uniplate* class requires no type system extensions, compared to rank-2 types for Scrap Your Boilerplate. The simplicity of the types required means that the user is free to concentrate on the operations within the class.

The central idea is to exploit a common property of many traversals: they only require value-specific behaviour for a single uniform type. In practical applications, this pattern is common. By focusing only on uniform type traversals, Uniplate is able to exploit well-developed techniques in list processing.

Uniplate defines various traversals, which are divided into two categories: queries and transformations. A query is a function that takes a value and extracts some information of a different type. A transformation takes a value and returns a modified version of the original value. All the traversals rely on the class *Uniplate*, which is shown below in a condensed form:

```

class Uniplate a where
  children  :: a → [a]
  contexts  :: a → [(a, a → a)]
  descend   :: (a → a) → a → a
  fold      :: (a → [r] → r) → a → r
  rewrite   :: (a → Maybe a) → a → a
  transform :: (a → a) → a → a
  universe  :: a → [a]

```

The Uniplate library provides a single method to support queries, the *universe* function. This function takes a data structure and returns a list of all structures of the same type found within it.

6 Generic programming libraries

For multi-parameter type traversals, to operate on data types within another data type, a second class called *Biplate* is provided. This feature needs the multi-parameter type classes extension. The function *universeBi*, provided by the library, is the multi type counterpart of *universe*:

$$\begin{aligned} \textit{universeBi} &:: \textit{Biplate} \ b \ a \Rightarrow b \rightarrow [a] \\ \textit{universeBi} &= \textit{universeOn} \ \textit{biplate} \end{aligned}$$

Furthermore, a separate tool (DERIVE) has been developed for automatically deriving instances of *Uniplate*. The tool is based on Template Haskell [41].

The next piece of code implements the *foldSal* test function using the Uniplate library:

$$\begin{aligned} \textit{foldSalUni} &:: \textit{Company} \rightarrow [\textit{Salary}] \\ \textit{foldSalUni} &= \textit{universeBi} \end{aligned}$$

Notice that the type annotation is mandatory.

Chapter 7

Criteria

Though the different generic programming libraries in Haskell pursue the same goal, they are all slightly different. We want to compare the libraries. For the comparison, we need a set of criteria, such as performance, expressivity and usability issues such as error messages. This chapter discusses criteria for comparing generic programming libraries in Haskell.

In the next chapter we introduce the *test functions* we use for evaluating and comparing. We discuss why and how those functions test the criteria introduced in this chapter.

Most previous work on generic programming focuses on either increasing the kind of scenarios in which generic programming can be applied (for example, Ralf Hinze’s work on polytypic values possess polykinded types [9] shows how to define generic functions of types of arbitrary kinds, instead of on types of a particular kind), or on obtaining the same kind of scenarios using fewer or no programming language extensions (for example, Generics for the Masses [11, 12] shows how to do a lot of generic programming without extending Haskell). Both goals are obtained by either inventing a new generic programming approach altogether, or by extending an existing approach. Each approach has a particular collection of scenarios it supports. This collection usually increases when a new approach is introduced.

We have studied a number of typical generic programming scenarios from the literature and we have also identified the features that are needed for each scenario. These features characterise generic programming from a user’s point of view, where a user is a programmer who *writes* generic programs. There are also users who only *use* generic programs (such as people that use **deriving** in Haskell), but the set of features needed by the latter kind of users is a subset of the set of features needed by the former kind of users. We translated the features to criteria. This chapter lists the criteria we have identified.

7 Criteria

| <i>Types</i> | <i>Expressiveness</i> | <i>Usability</i> |
|--|---|---|
| <ul style="list-style-type: none">• Full reflexivity• Views• Type universes• Intuition behind types• Multiple type arguments | <ul style="list-style-type: none">• First-class generic functions• Generic abstractions• Ad-hoc type cases• Ad-hoc constructor cases• Extensibility• Multiple arities• Local redefinitions• Consumers, transformers, ... | <ul style="list-style-type: none">• Performance• Portability• Amount of work per data type• Ease of learning |

Figure 7.1: Criteria structure

7.1 Organisation

The criteria are the most important input for the evaluation of the libraries and determine the output of this evaluation to a large extent. Therefore, the criteria need to be selected carefully, as unprejudiced as possible. We try to give a complete list of criteria, but criteria that are implied by other criteria are not listed, sometimes with a motivation. We have grouped criteria around the aspects of types, expressiveness and usability aspects. Figure 7.1 gives an overview of all the criteria and the way they are structured. The criteria originate from the following sources:

- new features introduced by existing approaches, such as polytypic values have polykinded types,
- Hinze’s et al. paper about Comparing approaches to generic programming in Haskell [13],
- the Haskell generics wiki page [37],
- our own ideas.

Although the list of criteria has been assembled with care, it will remain arbitrary to some extent. It may well be that the list is not exhaustive and that the criteria are not orthogonal and independent from each other. However, the criteria seem good enough to evaluate and compare the libraries.

7.2 Details

This section examines the criteria in more detail. When possible examples are given to illustrate their usage. How the criteria are covered by test

functions is described in Chapter 8. Some criteria are not covered by any test function. How those criteria are evaluated is explained in this section. Each criterion is accompanied with a set of scoring ‘rules’, when the ‘rules’ are omitted it defaults to:

| Grading | |
|-------------|--|
| <i>Good</i> | Satisfies the criterion. |
| <i>Bad</i> | Does <i>Not</i> satisfy the criterion. |

7.2.1 Types

The following criteria are used to see how generic programming libraries cope with a broad range of data types.

Full reflexivity. A generic programming library is fully reflexive if a generic function can be used on *any* Haskell data type. By design, Haskell has no reflection on data types, so a library has to be explicit about how data types are manipulated such that they are in the domain of generic functions, for example, by representing a data type as sum-of-products, or by using **deriving** *Data*. Different approaches allow different sets of data types to be manipulated.

For example, light-weight PolyP only supports regular data types with one parameter. Therefore light-weight PolyP does not score well on the full reflexivity criterion. The NOW library, however, is able to handle all the data types used in the tests (defined in Section 2.4), including higher-kinded data types and GADTs.

| Grading | |
|-------------------|--|
| <i>Good</i> | All Haskell data types can be used. |
| <i>Sufficient</i> | Most data types can be used, with exception of sophisticated ones (GADTs, higher-order kinded nested etc.) |
| <i>Bad</i> | Otherwise. |

Views. Does the library support more than one view on the structure of data types [18]? A view consists of a structure type and functions to get back and forth between an ordinary data type and its structure type. Besides the sum-of-products view, examples of views are: fixed-point and the spine view. Each view has its advantages and disadvantages.

An advantage of the spine view is that the function to transform back to the original data type is truly polymorphic. There is no need to define a function that transforms the structure data type back to the original type.

7 Criteria

A disadvantage is that the spine view¹ isn't able to support functions that produce data.

The evaluation of this criterion is based on the study of the papers that accompany the generic programming libraries.

Type universes. Some generic functions only make sense on a particular set of data types. Is it possible to define generic functions on a particular set of data types?

Consider a function that calculates the depth of a tree. We want to define such a function once and restrict its usage to tree-like data types. In our case the function should only work for *BinTree* and *WTree*. Calling the function with a data type that isn't element of the predefined set of allowed data types, should raise a *run-time* exception.

It is likely (but not necessarily) that type class based libraries can foresee in this criterion. Selectively instantiating data types allows us to restrict which data types are applicable.

Intuition behind types. Do types of generic functions in some way correspond to intuition? We consider a generic programming approach more intuitive if the type of a generic function is very similar to the type of a function instance that a programmer would write manually. We think that this similarity will make the generic type more understandable by the programmer and hence more intuitive.

Consider the type of the integer equality function: $Int \rightarrow Int \rightarrow Bool$. In EMGM, the representation for generic equality is passed as a type class context: $GRep\ Geq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. Thus, apart from the context, the type is also very similar to the equality instance function.

In SYB, generic equality is implemented by means of the geq' function which has type $(Data\ a1, Data\ a2) \Rightarrow a1 \rightarrow a2 \rightarrow Bool$. On a first reading this type is disconcerting, but necessary, because the traversal of the two terms is not done in parallel. This implies that the function does not enforce that the two arguments have the same type; in an other context this could be an advantage.

This criterion is extracted from the definitions of the test functions and the theory behind the library.

| Grading | |
|-------------|--|
| <i>Good</i> | For generic equality only constraints: $geq:: Ctx\ a \Rightarrow a \rightarrow a \rightarrow Bool$. |
| <i>Bad</i> | Otherwise. |

¹The normal spine view, not the lifted version

Multiple type arguments. Can a function be generic in more than one type argument? This criterion is satisfied if a generic programming library is able to define a generic *transpose* function. A generic transpose function, of type $transpose :: d (e a) \rightarrow Maybe e (d a)$, can be used to transpose, for example, a list of binary trees to a binary tree of lists.

7.2.2 Expressiveness

The next set of criteria is used to determine the expressiveness of the generic programming libraries.

First-class generic functions. Can a generic function take a generic function as an argument? An approach that supports first-class generic functions can be used to implement basic combinators that take generic functions as arguments. One example is the generic function *gmapQ* [29], it takes a generic function argument, applies it to all the fields of a constructor and returns a list with the results of the applications.

In LIGD a generic function is a polymorphic Haskell function, so it is a first-class value in Haskell implementations that support rank-2 polymorphism. The *gmapQ* function is definable in this library. It requires some sophisticated definitions, which are not trivial.

Generic abstractions. Is it possible to define generic functions just in terms of other generic functions? In other words, can we define generic functions without using case analysis on type representations?

For example combine two generic functions, say *foldSal* and a generic reduce, to create another *generic* function. In case of *foldSal* and *reduce* it is possible to create a function that collects all *Salary* values from a container data type, like a tree or list.

Ad-hoc type cases. Can a generic function contain specific behaviour for a particular data type and let the remaining data types be handled generically?

The *foldSal* function that lists all *Salary* values in a data type, indicates if a generic function can be given an ad-hoc definition for a data type. In the *foldSal* case, *Salary* is the ad-hoc data type.

Ad-hoc constructor cases. Can we give an ad-hoc definition for a particular constructor and let the remaining constructors be handled generically? E.g. ignore the *WithWeight* constructor of the *WTree Int Int* data type in a generic function that lists all integers.

7 Criteria

Extensibility. Can the programmer extend the *definition* of a generic function in a different module without the need for recompilation?

Consider the, in a library pre-defined, generic show function $gshow :: a \rightarrow String$. A generic programming library is extensible if we can extend this function so that it, for example, displays lists of characters in a different manner than other lists.

Multiple arities. The equality function can usually be defined in an approach to generic programming, but a generalisation of the function map on lists to arbitrary container types cannot be defined in all proposals.

The arity of a generic function is the number of active type arguments in its type. For example: the generic equality function $a \rightarrow a \rightarrow Bool$ has arity one, the generic map function $a \rightarrow b$ has arity two and the generic zip function $a \rightarrow b \rightarrow c$ has arity three.

The ‘generic functions of different arity’ criterion is probably not orthogonal with ‘multiple type arguments’. In other words, there is probably some overlap between them. We think nonetheless that the two give more information than when one or the other is discarded.

| Grading | |
|-------------------|------------|
| <i>Good</i> | Any arity. |
| <i>Sufficient</i> | Arity 2. |
| <i>Bad</i> | Arity 1. |

Local redefinitions. Can the programmer provide a custom function definition that overrides the default definition?

For instance when $collect$ is used on values of type $WTree\ Int\ Int$, it returns the empty list because that is the normal behaviour for $collect$ on Int values. But we can redefine $collect$ for the tree elements, i.e. the first argument of the $WTree$ type constructor, so that we collect the elements, ignoring the weights. In other words, we have a tree with a weight and a payload, both of type Int and we *only* want to collect the payload (the a in $WTree\ a\ w$).

Consumers, transformers and producers. Is the approach capable of defining consumer ($a \rightarrow T$), transformer ($a \rightarrow a$ or $a \rightarrow b$) and producer ($T \rightarrow a$) generic functions?

An example of a consumer is the generic show function $gshow :: a \rightarrow String$. The generic map is an example of a transformer $gmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$. The generic read $gread :: String \rightarrow a$ is a producer function, it takes a $String$ as input and parses it.

7.2.3 Usability

The following criteria are used to indicate the usability of a generic programming library.

Performance. Some proposals use many higher-order functions, others use conversions between data types and structure types. It is unknown how this affects performance. The performance should be measured along multiple dimensions, like Central Processing Unit (CPU)-time and memory usage. Performance measurements should be done with great care, the circumstances should be equal for each library.

We have done a preliminary CPU-time performance measurement, the results can be found in Section 9.4. Note that the outcome of the test is just an indication!

| Grading | |
|-------------------|---------------------|
| <i>Good</i> | Above average. |
| <i>Sufficient</i> | About average. |
| <i>Bad</i> | Well below average. |

Portability. Few proposals only use the Haskell98 standard, most use all kinds of (sometimes unimplemented) extensions to Haskell98, such as recursive type synonyms, multi-parameter type classes with functional dependencies, GADTs, etc. A proposal that uses few or no extensions is easier to port across different Haskell compilers.

The EMGM excels in portability since it can be defined entirely in Haskell98. In contrast to EMGM, RepLib uses more extensions and is more geared towards a specific Haskell compiler (GHC).

| Grading | |
|-------------------|---|
| <i>Good</i> | No extensions needed. |
| <i>Sufficient</i> | Only extensions available in most Haskell compilers (GHC, Hugs, Yhc). |
| <i>Bad</i> | Otherwise (including unimplemented extensions). |

Amount of work per data type. In some proposals generic functions automatically work for data types, in other proposals a user has to do some extra work per data type.

In SYB, data types must have the *Typeable* and *Data* instances. These can be generated by the GHC compiler, avoiding a lot of repetitive work for the programmer. Other libraries, like EMGM, leave it to the developer to implement the type representation (isomorphic functions).

7 Criteria

| Grading | |
|-------------------|---|
| <i>Good</i> | No extra work required (derivable). |
| <i>Sufficient</i> | Only one isomorphic function required (spine view). |
| <i>Bad</i> | Full embedded project pair definition required. |

Ease of learning. Some programming approaches are easier to learn than others.

For example, the essentials of the EMGM approach contain only a few lines of code. This is one of the reasons that it is quite easy to learn.

Chapter 8

Test functions

Chapter 6 made clear that there are many generic programming libraries and Chapter 5 stated the wish to develop a common generic programming library. A common generic programming library allows us to combine all the scattered effort and give lasting support. We introduced a set of criteria, which we will use to evaluate the libraries. Some criteria are best tested using a generic function that can only be defined if a criterion is satisfied. For example the criterion ‘Multiple arities’ is satisfied to a certain degree if the generic *map*¹ function can be constructed.

In this chapter we introduce a number of *generic* functions, that will be used to examine to what extent a library is able to satisfy some of the criteria. These functions are derived from scenarios. A scenario is a particular example usage of generic programming. The generic functions in this chapter are defined using the ‘Extensible and Modular Generics for the Masses’ (EMGM) [6] library.

We have implemented most of these functions for each library, if possible. We use these implementations together with the underlying theory for each library in the evaluation chapter to evaluate the libraries.

We use the data types defined in Section 2.4 to exemplify the generic functions. In order to be complete, the equality and *FoldTree* test are repeated here in a condensed form.

8.1 Equality

The equality function takes two arguments and determines whether or not they are equal. At each step, the top-level constructors of the arguments are compared for equality and, if they are equal, the children of the constructors are pairwise compared, recursively. If the top-level constructors are constructors from primitive types such as *Int* and *Char*, we use the

¹The *map* function on list generalised to arbitrary container data types, such as binary trees.

8 Test functions

equality function on that primitive type. Both the structure and the ‘content’ of the arguments are compared. The generic equality function has the type $Geq\ a\ b\ c$, which is defined by:

newtype $Geq\ a\ b\ c = Geq\ \{geq' :: a \rightarrow a \rightarrow Bool\}$.

Generic equality is of arity one, so the b and c generic type variables are discarded. A generic function is implemented by providing an instance of the class *Generic* for this type:

instance *Generic* Geq **where**

| | | |
|---------------------------|---|---|
| <i>unit</i> | = | $Geq\ (\lambda x\ y \rightarrow \mathbf{case}\ (x, y)\ \mathbf{of}$ |
| | | $(Unit, Unit) \rightarrow True$ |
| <i>plus\ a\ b</i> | = | $Geq\ (\lambda x\ y \rightarrow \mathbf{case}\ (x, y)\ \mathbf{of}$ |
| | | $(Inl\ xl, Inl\ yl) \rightarrow geq'\ a\ xl\ yl$ |
| | | $(Inr\ xr, Inr\ yr) \rightarrow geq'\ b\ xr\ yr$ |
| | | $_ \rightarrow False$) |
| <i>prod\ a\ b</i> | = | $Geq\ (\lambda x\ y \rightarrow geq'\ a\ (outl\ x)\ (outl\ y) \wedge$ |
| | | $geq'\ b\ (outr\ x)\ (outr\ y))$ |
| <i>view\ iso\ _ _ a</i> | = | $Geq\ (\lambda x\ y \rightarrow geq'\ a\ (from\ iso\ x)\ (from\ iso\ y))$ |
| <i>char</i> | = | $Geq\ (\lambda x\ y \rightarrow x \equiv y)$ |
| <i>int</i> | = | $Geq\ (\lambda x\ y \rightarrow x \equiv y)$ |
| <i>float</i> | = | $Geq\ (\lambda x\ y \rightarrow x \equiv y)$ |

instance *GenericCompany* Geq .

Notice the last instance declaration. The *GenericCompany* class is a subclass of *Generic* and makes it possible to make ad-hoc cases available for every data type within the compound data type *Company*. Every data type within the *Company* data type is added as a method to the subclass *GenericCompany* along with a default implementation. This default implementation uses the view (isomorphism) method. If we want to add an ad-hoc case to the generic function, the default implementation can be overridden.

In the case that a compound data type is extended with an additional data type, the subclass has to be extended with a method and a default implementation. The generic functions remain unchanged. In contrast to one subclass, we can make a subclass for every individual data type. In this case, adding a data type would mean that *every* function has to add an instance declaration, which is tedious and error prone.

To use the generic equality function on a particular data type, we select the geq' method from Geq by using the type representation for the data type:

$equalCompany :: Company \rightarrow Company \rightarrow Bool$
 $equalCompany = geq'\ over.$

8.1.1 Generic Rose Trees

This test applies the equality function on generic rose trees. Generic rose trees, as defined in Section 2.4, is used to test whether a library is able to handle data types with a so-called *second-order* kind, where the order of a kind is given by:

$$\begin{aligned} \text{order } (\star) &= 0 \\ \text{order } (\kappa \rightarrow \nu) &= \max \{1 + \text{order } (\kappa), \text{order } (\nu)\}. \end{aligned}$$

The definition of the test function:

$$\begin{aligned} \text{equalGRoseListInt} &:: \text{GRose [] Int} \rightarrow \text{GRose [] Int} \rightarrow \text{Bool} \\ \text{equalGRoseListInt} &= \text{geq}' \text{ over}. \end{aligned}$$

isn't that exciting. The actual work is in the type representation of the generic rose trees. However, the type representation is surprisingly straightforward to do in EMGM.

8.1.2 Generic nested rose trees

It is very hard or even impossible to represent a higher-order kinded nested data type in EMGM. The test function, that compares two generic nested rose tree values for equality, cannot be defined in EMGM. There is no straightforward way to represent **data** *Two* $f\ a = \text{Two } (f\ (f\ a))$, which is part of the *NGRose* data type.

We want to stress that we are *not* claiming that it is not possible, there may be clever, tricky way to get it working after all.

8.1.3 Trees with weights

The function *equalWTree* implements the equality function for values *WTree*. However, the implementation should be done generic with an *ad-hoc constructor* case (for *WithWeight*). It is possible to strip of the weights:

$$\begin{aligned} \text{equalWTree} &:: \text{WTree Int Int} \rightarrow \text{WTree Int Int} \rightarrow \text{Bool} \\ \text{equalWTree } (\text{WithWeight } t_1 _) &(\text{WithWeight } t_2 _) = \text{equalWTree } t_1\ t_2 \\ \text{equalWTree } x\ y &= \text{geq}' \text{ over } x\ y \end{aligned}$$

alas, the representation function calls itself recursively. This does not work; it is possible to cheat and adapt the dispatcher to a function that compares *WTree* values. Normally the dispatcher, calls the correct generic class method.

8.2 Reduce

The *reduce* function is a higher-order function that traverses a data type value in some order and constructs a return value. For constructing the return value, *reduce* takes a function for combining two arguments and a start value. It is a bit similar to Haskell's *foldr* function, which takes a binary function, a base value and a list as input and returns the value obtained by applying the function recursively, using the start value for the empty list and the binary function for non-empty lists. For example: folding a list [1,2,3,4] with the addition operator and base value zero, yields 10.

The generic *reduce* function can reduce values from many data types. It collects elements from a value and combines them with the given function. The type of the function is given by:

```
newtype Red c a b d = Red { red' :: (c → c → c) → c → a → c }.
```

Notice that the generic type variables *b* and *d* aren't used. The *c* is an ordinary polymorphic type variable, in contrast to *a*, *b* and *d*, which are *generic* type variables.

```
instance Generic (Red c) where
  unit           = Red (λ_ e _ → e)
  plus a b      = Red (λop e x → case x of
                        Inl l → red' a op e l
                        Inr r → red' b op e r)
  prod a b      = Red (λop e x → (red' a op e (outl x))
                        'op' (red' b op e (outr x)))
  view iso1 iso2 _ a = Red (λop e x → red' a op e (from iso1 x))
  char          = Red (λ_ e _ → e)
  int           = Red (λ_ e _ → e)
  float        = Red (λ_ e _ → e)
```

```
defValue v = Red (λ_ _ _ → v)
```

Generic *reduce* combines the type-*c* values in a value of type *a* using the 'combine' ($c \rightarrow c \rightarrow c$) function given as an argument. A *local redefinition* is necessary to get the *c* values out of *a*, otherwise only the base value is combined. The function *defValue* foresees in the local redefinition need.

Function *collect* collects type-*c* values in a data type of kind $\star \rightarrow \star$ containing *c*-values in a list. It takes as argument a function that tells where the *c*-values occur:

```
collect :: (Red [a] a b d → Red [a] a1 b1 d1) → a1 → [a]
collect rep = red' (rep (Red (λ_ _ x → [x]))) (++) [].
```

For example, if *tree* is the type representation function for the data type *WTree a w*, we can collect the *a*-values:

```
collectListTree :: [WTree a w] → [a]
collectListTree = collect (λrep → rList (tree rep (defValue []))).
```

8.2.1 Nested data types

The *collect* function is also used to test if the library can handle nested data types. The *collectPerfect* function collects all *a*-s from a nested data type *Perfect*:

```
collectPerfect :: Perfect a → [a]
collectPerfect = collect perfecttree.
```

Where *perfecttree* is the type representation of the *Perfect* data type.

8.3 Generic Map

The *gmap* function is a higher-order function that takes a function and a structure of elements and applies the function to all elements in the structure. It abstracts from the standard *map* function on lists. The following code listing, shows how to implement it:

```
newtype Gmap a b c = Gmap { gmap' :: a → b }

instance Generic Gmap where
  unit           = Gmap (λx → x)
  plus a b      = Gmap (λx → case x of
                        Inl l → Inl (gmap' a l)
                        Inr r → Inr (gmap' b r))
  prod a b      = Gmap (λx → (gmap' a (outl x)) ×
                        (gmap' b (outr x)))
  view iso1 iso2 _ a = Gmap (λx → to i2 (gmap' a (from i1 x)))
  char          = Gmap (λx → x)
  int           = Gmap (λx → x)
  float        = Gmap (λx → x).
```

For example, we can define the original *map* function on lists, as well as any other instance of the *FunctorRep* class, as:

```
class FunctorRep g f where
  functorRep :: g a1 a2 a3 → g (f a1) (f a2) (f a3)
```

8 Test functions

```
gmap :: FunctorRep Gmap f ⇒ (a → b) → f a → f b  
gmap f = gmap' (functorRep (Gmap f)).
```

Here *functorRep* is the functor representation of lists. A functor representation is similar to a type representation, in which the type argument is explicitly represented. So a functor representation has kind $\star \rightarrow \star$. In EMGM, the functor representation of lists is the instance of lists of the class *FunctorRep* defined by:

```
class Generic g ⇒ FunctorRep g f where  
  functorRep :: g a b c → g (f a) (f b) (f c)
```

Another function that is defined in the test:

```
mapListBTree :: (a → b) → [BinTree a] → [BinTree b]  
mapListBTree f = gmap' (rList (bintree (Gmap f)))
```

is used to apply a function on the *a*-s in a list of binary trees.

8.3.1 GMapQ

An approach that supports first-class generic functions can be used to implement more basic combinators that take generic functions as arguments. One example is the generic function *gmapQ*, it takes a generic function argument, applies it to all the fields of a constructor, and returns a list with the results of the applications.

In EMGM *gmapQ* seems only possible with a lot of trickery, and cannot be defined in a straightforward manner. We omit the current implementation because it is not stable.

8.4 FoldInt

The *foldInt* test collects all occurrences of *Int* values in a data type. For example, we might want to collect all *Int* values that appear in a value of the *BinTree* data type:

```
newtype FoldInt a b c = FoldInt {foldInt' :: a → [Int]}
```

```
foldInt :: GRep FoldInt a ⇒ a → [Int]  
foldInt = foldInt' over.
```

8.5 Generic Show

The *gshow* function takes a value of a data type as input and returns its representation as a string. It can be viewed as the implementation of **deriving Show** in Haskell. The definition of the generic show *gshows* function:

```
newtype Gshows a b c = Gshows { gshows' :: a → ShowS }
```

```
instance Generic Gshows where
```

```
  unit          = Gshows (λx → showString "")
  plus a b      = Gshows (λx → case x of
                        Inl l → gshows' a l
                        Inr r → gshows' b r)
  prod a b      = Gshows (λx → gshows' a (outl x).
                        showString " ".
                        gshows' b (outr x))
  view iso _ _ a = Gshows (λx → gshows' a (from iso x))
  char          = Gshows (λx → shows x)
  int           = Gshows (λx → shows x)
  float        = Gshows (λx → shows x)
  constr n ar a = Gshows (λx → if ar ≡ 0 then
                        showString (n)
                        else
                        showChar ' (.
                        showString (n).
                        showChar ' '.
                        (gshows' a (x)).
                        showChar ' )')
```

```
instance GenericCompany Gshows
```

```
  gshows :: GRep Gshows a ⇒ a → ShowS
  gshows = gshows' over.
```

There are some interesting aspects to the *gshows* function. First, it has a number of ad-hoc cases for showing values of particular data types in a different way. For example, constructors of arity one are shown without parentheses. Here is the definition of the *gshows* function on *Company* data types:

```
  gshowsCompany :: Company → String
  gshowsCompany x = gshows x "".
```

8.5.1 GADT

EMGM doesn't support type representations of GADTs.

The test uses the *gshows* function to print a GADT value. There are very few libraries that are able to handle GADTs. RepLib and NOW are able to do so.

8.6 Increase

Suppose a manager wants to give all his personnel a raise. We use a generic function *increase* to raise the salaries in a value of an arbitrary data type (even values of data types that do not contain salaries). The instance of the generic function on the data type *Company* type takes a *Float* and a *Company* value as input and returns the same value with all values of type *Salary* updated with an amount related to the first argument of type *Float*. The definition of the *increase* function:

```
newtype Ginc a b c = Ginc { ginc' :: Float → a → a }
```

```
instance Generic Ginc where
```

```
  unit      = Ginc (λf x → x)
```

```
  plus a b  = Ginc (λf x → case x of
```

```
    Inl l → Inl (ginc' a f l)
```

```
    Inr r → Inr (ginc' b f r))
```

```
  prod a b  = Ginc (λf x → (ginc' a f (outl x)) ×  
    (ginc' b f (outr x)))
```

```
  view iso _ _ a = Ginc (λf x → to iso (ginc' a f (from iso x)))
```

```
  int      = Ginc (λf x → x)
```

```
  char     = Ginc (λf x → x)
```

```
  float    = Ginc (λf x → x)
```

```
instance GenericCompany Ginc where
```

```
  salary    = Ginc (λf x → incS f x).
```

The *incS* function increases a *Salary* value with a factor *k*:

```
incS :: Float → Salary → Salary
```

```
incS k (S s) = S (s * (1 + k)).
```

The *increase* function restricted to *Company* data types:

```
increase :: Float → Company → Company
```

```
increase = ginc' over.
```

Abstracting from salaries and companies, an 'increase' function adapts occurrences of values from a particular data type in another data type.

8.7 Generic transpose

The generic transpose function takes a value of type d ($e a$) and transposes it to a value of type e ($d a$). Implementing this function using EMGM, or any other library, is far from trivial. Below is an excerpt from the full implementation. The following function is the less generic transpose function; d is fixed to the list type:

```
listTrans :: (FunctorRep Gmap e, FunctorRep GzipWith e) =>
  [e a] -> e [a]
listTrans (x : []) = gmap (\s -> [s]) x
listTrans (x : xs) = zipWithFunctor (:) x (listTrans xs).
```

The `listTrans` function uses the generic `zipWith` function, which has the following type:

```
zipWithFunctor :: FunctorRep GzipWith f => (a -> b -> c) ->
  f a -> f b -> f c.
```

An example usage of transposing a list of binary trees to a binary tree of lists:

```
> listTranspose [Bin (Bin (Leaf 32) (Leaf 3)) (Leaf 4),
  Bin (Bin (Leaf 32) (Leaf 3)) (Leaf 4)]
Bin (Bin (Leaf [32, 32]) (Leaf [3, 3])) (Leaf [4, 4]).
```

The full implementation of `gtranspose` should be even more generic. In the current definition it is not possible to specify which container type should be transposed, in the case there are more than two. Suppose we have a data type $T_1 (T_2 (T_3 a))$, we want to *specify* which data types should be transposed. We have three possibilities: $T_2 (T_1 (T_3 a))$, $T_3 (T_2 (T_1 a))$ or $T_1 (T_3 (T_2 a))$. To be able to specify the container types, we need *two* type arguments. The implementation of the more generic version is postponed due to other priorities.

8.8 Performance test

We use the `bigeq` function to test the performance of a library. The `bigeq` function checks a large tree for equality. The definition of `bigeq`:

```
bigeq :: Int -> Bool
bigeq n = equalBinTreeChar t t
  where t = fulltree bintree n ' * '
```

where `fulltree` is defined in Section 8.10. The generic producer `fulltree` produces a `BinTree` with depth n and `' * '` in the leaves.

8 Test functions

The *equalBinTreeChar* uses the generic *geq'* function:

```
equalBinTreeChar :: BinTree Char → BinTree Char → Bool
equalBinTreeChar = geq' over.
```

The actual performance test routine calculates the time cost of the *bigeq* function.

8.9 Generic lookup

The *lookup* function from the Prelude has type $Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$ and returns the value belonging to the given key, protected by the *Maybe* monad. The generic lookup function *glookup* replaces the constraint $Eq\ a$ with $GRep\ Geq\ a$, which means that the function works for all *a*-s, not only on instances of the *Eq* class. The generic lookup function can be defined as follows:

```
glookup :: (GRep Geq a) => a → [(a, b)] → Maybe b
glookup key ((x, y) : xys) | geq' over key x = Just y
                          | otherwise       = glookup key xys
glookup key ([] = Nothing.
```

A lookup function to get the ‘burger service nummer²’ making use of *glookup*:

```
lookupBsn :: Person → [(Person, Int)] → Maybe Int
lookupBsn = glookup
```

The generic lookup function tests whether a library is able to satisfy the ‘Generic abstraction’ criterion.

8.10 Fulltree

The capability of constructing a generic read function is tested by means of the generic *fulltree* function. This function creates a value of a specified type with size *s*. The actual size depends on the way the type representation is done. In case of the *BinTree* type, *s* represents the depth of the tree. The *fulltree* function is defined as follows:

```
newtype FullTree d a b c = FullTree { fulltree' :: Int → d → a }
```

```
instance Generic (FullTree d) where
  unit          = FullTree (λd _ → Unit)
  plus a b      = FullTree (λd x → if d > 0 then
```

²The Dutch social security number.

```

                                Inr (fulltree' b d x)
                                else
                                Inl (fulltree' a d x)
prod a b      = FullTree (\d x → (fulltree' a (d - 1) x) ×
                                (fulltree' b (d - 1) x))
view iso _ _ a = FullTree (\d x → to iso (fulltree' a d x))
char          = FullTree (\d _ → 'a')
int           = FullTree (\d _ → 1)
float        = FullTree (\d _ → 1.0).

```

The first argument of *fulltree* is a container type representation, like *rList* and *binTree*:

```

fulltree :: (FullTree d d b c → FullTree d1 a b1 c1) → Int → d1 → a
fulltree rep s = fulltree' (rep (FullTree (\_x → x))) s.

```

Some example usages:

```

> fulltree binTree 2 "hi"
Bin (Bin (Leaf "hi") (Leaf "hi")) (Bin (Leaf "hi") (Leaf "hi"))
> fulltree rList 6 "hi"
["hi", "hi", "hi", "hi", "hi", "hi"].

```

This function is used in the performance test, Section 9.4, to generate a large tree.

8.11 Test suite

The test suite consists of a set of Haskell programs that test the generic functions as described in the previous sections. Each test consists of a module, written by the ‘end user’ using the library under test, from which a generic function is called on an value of a predefined data type (like *Company*). This Haskell module, which may be found at the root directory, e.g. `TestGEq.hs`, imports a generic module. It is this generic module that contains the generic function, already specialised for the requested data type.

Running tests. The test suite can be obtained using darcs³:

```
darcs get http://darcs.haskell.org/generics
```

Change the directory to the `comparison` subdirectory and type

```
runghc test.hs [lib...]
```

on the command line to run the test on all libraries. The optional command line parameter (`[lib..]`) can be used to select one or more libraries to test. To use `runghc` we need of course the GHC compiler.

³A free distributed version control system, <http://darcs.net>.

8 Test functions

| | <i>Equality</i> | <i>Reduce</i> | <i>Map</i> | <i>FoldInt</i> | <i>Show</i> | <i>Increase</i> | <i>Transpose</i> | <i>Efficiency</i> | <i>Lookup</i> | <i>Fulltree</i> |
|--------------------------|-----------------|---------------|------------|----------------|-------------|-----------------|------------------|-------------------|---------------|-----------------|
| Full reflexivity | ● | | | | | | | | | |
| Multiple type args | ◐ | | | | | | ● | | | |
| First-class | | | ● | | | | | | | |
| Generic abstr. | | | | | | | | | ● | |
| Ad-hoc type cases | | | | ● | ● | | | | | |
| Ad-hoc cons. cases | ● | | | | | | | | | |
| Extensibility | ● | | | | | | | | | |
| Multiple arities | | | ● | | | | ● | | | |
| Local redefinitions | | ● | ● | | | | | | | |
| Cons., prod., ... | | | ● | ● | ● | ● | | | | ● |
| Performance | | | | | | | | ● | | |
| <i>Fully covered</i> | ● | | | | | | | | | |
| <i>Partially covered</i> | ◐ | | | | | | | | | |

Table 8.1: Functions set out against criteria.

8.12 Criteria coverage

Some criteria are not tested by any example, e.g. ease of learning. Other criteria are tested by *all* functions, e.g. amount of work per data type. And some other are not easily testable, e.g. generic transpose. Table 9.1 gives an overview of how the test cover the criteria. The following criteria are not tested by means of a generic test function:

- views,
- type universes,
- intuition behind types,
- portability,
- ease of learning.

These criteria are evaluated based on the theory of the libraries, as described in the corresponding papers.

Part III
Evaluation

Chapter 9

Evaluation

The previous chapters introduce a framework with which we can evaluate generic programming libraries in Haskell. Due to time constraints not all libraries could be evaluated. Moreover, evaluating a generic programming library requires expert knowledge about it. In the ideal situation the library developer should do the evaluation. We gave the development of the criteria together with the test the highest priority. The evaluation framework, the criteria together with the test suite, is the main result of our work. Since libraries change over the years, evaluations of libraries are less stable. Libraries evolve, new versions and new libraries will appear every now and then.

Chapter 6 divides the libraries into two distinct groups. We have chosen two representatives from each group to evaluate. The EMGM library has been selected from the light-weight approaches group, because we have used it for our examples throughout the document. The SYB revolutions serves as the representative for the strategic programming group. It is one of the most recent libraries based on the original SYB approach. For the remaining libraries we give a short evaluation, highlighting the most prominent features.

Evaluating and comparing generic programming libraries is difficult due to a large number of influencing factors, many of which are hard to express. The outcome of some evaluations is therefore a subjective conclusion which depends on the ones who state them. Moreover, a discussion about the outcome of most evaluations does not often converge because of different priorities given by different people which are often not explicitly formulated and agreed upon. One person might find that library *A* is better than library *B* because of its performance results, while another person might emphasise the fact that library *B* is far more portable than *A*. Although it is difficult to express these issues, transparency and explicitness are necessary in order to make decisions in a library comparison.

9 Evaluation

Evaluation procedure. The evaluation of a library consists of a few steps. First, all the test functions are implemented using the library, if possible. Second the underlying theory, described in the corresponding paper, is studied in detail. Using the knowledge from the previous steps the criteria are evaluated according to the grading rules.

9.1 EMGM

All of the implementations of test functions are given in Chapter 8. These form the basis of the evaluation. The next paragraphs give the score of the EMGM library for each criterion. The outcome of the evaluation is summarised in Table 9.1.

Full reflexivity. In EMGM higher-kinded data types are representable. Nested data types can be given a type representation in EMGM using polymorphic recursion, which is supported in Haskell98. EMGM is almost full reflexive, were it not the case that higher-order kinded nested data types and GADTs are very hard to represent (the *NGRose* data type). Score: ‘sufficient’.

Views. EMGM supports only the sum-of-products view. Score: ‘bad’.

Type universes. Class-based generic libraries can restrict the applicable types (universe) by means of instantiation. A data type that is not instantiated for a particular generic function cannot be used and will give a compile-time error if we try to apply it. Score: ‘good’.

Intuition behind types. In EMGM, the representation for generic equality is passed as a type class context: $GRep\ Geq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. Thus, apart from the context, the type is very similar to the equality instance function. Score: ‘good’.

Multiple type arguments. EMGM supports multiple type arguments. A first version of the generic transpose has been implemented using EMGM. Score: ‘good’.

First-class generic functions. In EMGM *gmapQ* has the following type signature:

$$\begin{aligned} gmapQ &:: GRep\ (GMapQ\ g)\ a \\ &\Rightarrow (\forall a.g\ a \rightarrow a \rightarrow b) \rightarrow a \rightarrow [b]. \end{aligned}$$

This generic mapping function supports extensible generic function arguments. However, this requires a small modification to how this approach

represents data types. In summary, this approach supports higher-order generic functions but their use requires rather sophisticated definitions. For this reason we have chosen to not give it the best score for this criterion, score: ‘sufficient’.

Generic abstractions. In EMGM, we can abstract over the type argument of a generic function, for instance the *geq'* in the *glookup* implementation of the generic lookup test. It is necessary to use the context *GRep Geq a*. Score: ‘good’.

Ad-hoc type cases. In EMGM, ad-hoc definitions are obtained by subclassing *Generic* and give specific class methods for the ad-hoc type case. Score: ‘good’.

Ad-hoc constructor cases. In EMGM it is possible to add specific behaviour for a certain constructor. However, the scheme used is not extensible, e.g. suppose the generic equality function is predefined, then it is not possible anymore to access the constructor case. The rating is therefore ‘sufficient’.

Extensibility. EMGM generic functions are extensible because ad-hoc definitions are provided using instances of subclasses of *Generic*, which can reside in different modules.

Multiple arities. EMGM has support for functions up to arity three (like the *gZipWith* function). Functions of lower arity can be defined with the same library, at the expense of some cluttering in the type signatures. Score: ‘good’.

Local redefinitions. The reduce function can be implemented in EMGM. The EMGM library supports local redefinitions. Score: ‘good’.

Consumers, transformers and producers. The EMGM library supports functions that consume, transform and produce values of data types. Score: ‘good’.

Performance. Considering the preliminary performance test and grading rules, EMGM scores: ‘good’.

Portability. One of the key features of the EMGM library is that it can be defined completely in Haskell98. If we want to use the generic dispatcher, which is very convenient, multi-parameter type classes are needed. EMGM scores: ‘good’.

Amount of work per data type. As we have seen in the previous chapters, a full embedding projection pair has to be defined for every data type. This only has to be done only once and can be used for all generic functions, but still it is some work. Following the rules, EMGM scores: ‘bad’.

Ease of learning. The essentials of the EMGM approach contain only a few lines of code. This is one of the reasons that is quite easy to learn. Though it requires some knowledge about type classes. Score: ‘good’.

9.2 SYB Revolutions

SYB uses the so-called spine view on data types. An advantage of the spine view is its generality: it is applicable to a large class of data types, including generalised algebraic data types. Its main weakness roots in the orientation on values: one can only define generic functions that consume data but not ones that produce data. Furthermore, functions that abstract over type constructors are out of reach. This deficiency is solved by SYB Revolutions it introduces the ‘type spine’ view for defining generic producers and the ‘lifted spine’ view, which makes it possible to define generic functions that abstract over type constructors.

Full reflexivity. The spine view treats data uniformly as constructor applications; it is, in a sense, value-oriented. The SYB Revolutions supports a large class of data types and scores: ‘good’.

Views. SYB Revolutions supports more than one view, next to the spine view, the ‘type spine’ and ‘lifted spine’ view are supplied. However, the extra views follow the same structure as the original spine view and could be seen as one view consisting of ‘sub views’. The rating is therefore: ‘bad’.

Type universes. There is no way to limit the universe of a generic function. Score: ‘bad’.

Intuition behind types. In SYB Revolutions generic equality takes one additional argument: the representation of the type of values that are to be compared. Score: ‘sufficient’.

Multiple type arguments. From a theoretical point of view, SYB Revolutions supports multiple type arguments. This remains to be validated by the implementation of the test function *gtranspose*.

First-class generic functions. In SYB Revolutions a generic function is a polymorphic Haskell function, so they are first-class values in Haskell implementations that support rank-2 polymorphism. It follows that *gmapQ* is definable in these approaches. Score: ‘good’.

Generic abstractions. The run-time representations of data types can be passed as arguments to the caller of a generic function to make a generic abstraction. Score: ‘good’.

Ad-hoc type and constructor cases. The representation data type is fixed in the library module. So a type-specific case for a data type that is not represented cannot be provided, if the representation data type is not extended and the library recompiled.

An alternative to give an ad-hoc definition is to pattern match the structure representation of the ad-hoc type, and inspect the structure value. However this is a very tiresome way to provide ad-hoc definitions. Score: ‘bad’.

Extensibility. Generic functions are not extensible because it is impossible to add a function case in a different module from where the function is defined in Haskell. Score: ‘bad’.

Multiple arities. In SYB Revolutions multiple arities mean several data type declarations for each type representation. Score: ‘good’.

Local redefinitions. The lifted type representation supports local redefinitions using an additional *Id* constructor that dictates where in the type to apply the redefinition. Score: ‘good’.

Consumers, transformers and producers. The library scores very good on this criterion. We are able to define, the *gmap*, *gshow* and a producer function using the SYB Revolutions library.

Performance. In the performance test, the SYB Revolutions library scores just below average.

Portability. SYB Revolutions needs the `-fglasgow-exts` extension, and scores: ‘sufficient’.

Amount of work per data type. A lifted version of the spine view must be used for generic functions with arity 2 and higher. This lifted view requires that every represented data type has a ‘lifted’ counterpart and conversion functions to and from this lifted data type. This implies

9 Evaluation

that functions of higher arity require additional work from the programmer, compared to ordinary type representations for arity one functions. Score: ‘bad’.

Ease of learning. It is quite easy to understand how to define generic functions and type representations, due to the fact that the type structure is reflected onto the value level. Score: ‘good’.

9.3 Summary remaining libraries

In this section we only highlight the salient features of the libraries: where do they excel and where do they fail. Table 9.1 gives an overview of the evaluation of all libraries. Bear in mind that the table displays only a *preliminary* evaluation. Although some more tests need to be performed, this preliminary evaluation gives already a good indication.

LIGD. Generic functions in LIGD can be applied to higher-kinded data types.

In LIGD generic functions and their type arguments are encoded as Haskell functions and representation values, which are first-class in Haskell. Therefore LIGD supports first-class functions and generic abstractions. The downside is that Haskell functions are not extensible, so generic functions are not extensible either.

The implementation of LIGD only requires Haskell98 and existential types.

In LIGD the representation data type is fixed in the library module. So a type-specific case for a data type that is not represented cannot be provided, if the representation data type is not extended and the library recompiled.

Scrap Your Boilerplate with class. SYB normally uses instances derived by the GHC compiler, but type class deriving is not supported for higher-kinded data types.

In SYB, generic equality is implemented by means of the *geq'* function which has type $(Data\ a1, Data\ a2) \Rightarrow a1 \rightarrow a2 \rightarrow Bool$. On a first reading this type is disconcerting, but necessary, because the traversal of the two terms is not done in parallel. This implies that the function does not enforce that the two arguments have the same type. In a different context the two different types might be an advantage.

In SYB, generic abstractions can be defined by leaving the type arguments of generic functions abstract but constrained by the *Data* and *Typeable* type classes.

RepLib. RepLib uses a GADT to represent types and a type class to support convenient access to them.

RepLib can restrict the universe of types of a generic function, by only instantiating those types for a specific function.

RepLib can derive instances for the type representation.

Light-weight PolyP. Light-weight PolyP is limited to regular data types (with one parameter) and cannot handle mutually recursive data types, so the set of data types (the universe) supported is relatively small. It is however possible to define functions generic in two type parameters like *gtranspose* :: ... $\Rightarrow d (e a) \rightarrow e (d a)$.

The limited universe means that PolyP is not suitable as a general generic library. It is included here as a ‘classic reference’ and because of its expressivity.

Derivable type classes. In DTCs, generic functions are tied to class methods. However, type classes are not first-class citizens. Consequently, generic functions are not first class either.

Generic programming, now! In NOW the data type of type representations is an *open* GADT. A fundamental problem with this approach is that open data types and open functions are not supported in Haskell or one of its extensions.

The NOW library supports multiple views on data types: (variants of) the spine view, but also the sum-of-products view.

Smash Your Boilerplate. In Smash, strategies are extensible; programmers may define their own traversal strategy and instantiate *LDat* for specific data types with regards to that new strategy.

Uniplate. Uniplate has been tested only superficially.

9.4 Performance

The performance test is part of the test suite. The generic programming libraries taking part of the efficiency test implement a certain test function (Section 8.10). The calculation of the test function is timed. A makefile is used to invoke the performance test:

```
make time
```

The table below show the results of the preliminary performance test. This particular test run is performed on a Intel Core2Duo 2.0 Giga Hertz (GHz) computer with 2 Giga Byte (GB) of memory, running Mac OS X.

9 Evaluation

| Preliminary speed test (result in ms) | |
|---------------------------------------|------|
| <i>Replib</i> | 132 |
| <i>PolyP</i> | 212 |
| <i>EMGM</i> | 296 |
| <i>LIGD</i> | 396 |
| <i>SYB Rev.</i> | 512 |
| <i>SYB</i> | 1020 |

| | <i>LIGD</i> | <i>SYB Rev.</i> | <i>EMGM</i> | <i>DTCs</i> | <i>SYB</i> | <i>RepLib</i> | <i>Smash</i> | <i>NOW</i> | <i>Uni</i> | <i>PolyP</i> |
|----------------------|-------------|-----------------|-------------|-------------|------------|---------------|--------------|------------|------------|--------------|
| Full reflexivity | ● | ● | ● | - | ● | ● | - | ● | - | ○ |
| Views | ○ | ○ | ○ | - | - | ○ | ● | ● | ○ | ○ |
| Type universes | ○ | ○ | ● | - | - | ● | - | ○ | - | ● |
| Intuition | ● | ● | ● | - | ● | ● | - | ● | - | ● |
| Multiple type arg. | - | - | ● | - | - | - | - | - | - | ● |
| First-class | ● | ● | ● | - | ● | ● | - | - | - | ○ |
| Generic abstractions | ● | ● | ● | - | ● | - | - | - | - | ● |
| Ad-hoc type cases | ○ | ○ | ● | ● | ● | ● | ● | ● | - | ● |
| Ad-hoc cons. cases | ○ | ○ | ● | - | ● | ● | ● | ● | - | ○ |
| Extensibility | ○ | ○ | ● | - | ○ | - | ● | - | - | ○ |
| Multiple arities | ● | ● | ● | - | ○ | ● | - | ● | - | ○ |
| Local redefinitions | ● | ● | ● | - | ○ | - | - | ● | ● | ● |
| Cons., Trans., ... | ● | ● | ● | - | ● | ● | ● | ● | - | ● |
| Performance | ● | ● | ● | - | ○ | ● | - | - | - | ● |
| Portability | ● | ● | ● | - | ● | ○ | - | ○ | ● | ● |
| Amount of work | ○ | ○ | ○ | - | ● | ● | - | ● | ● | ○ |
| Ease of learning | - | ● | ● | - | - | ○ | - | ● | - | ● |
| <i>Good:</i> | ● | | | | | | | | | |
| <i>Sufficient:</i> | ● | | | | | | | | | |
| <i>Bad:</i> | ○ | | | | | | | | | |
| <i>Undetermined:</i> | - | | | | | | | | | |

Table 9.1: Evaluation of generic programming approaches

Chapter 10

Epilogue

This master’s thesis gives a brief introduction into functional programming, advanced features of the Glasgow Haskell compiler and generic programming. The library ‘Extensible and Modular Generics for the Masses’ is used as an example library throughout the entire document. This thesis provides the background details of the library along with test functions/scenarios implemented using the library.

We want to offer (potential) users of generic programming a stable development platform. We identified the need for a *common generic programming library* in Haskell. This master’s thesis takes the first step towards a common library, we define a set of well documented criteria. Together with the criteria, a set of test functions and a test procedure make the evaluation framework complete.

To reach the stated goal of a common generic programming library, research questions are posed in Section 5.2, with its main question: how can generic programming libraries be evaluated? This master’s thesis answers that question by giving a set of criteria and a test suite. Furthermore, the use of the criteria and test suite are demonstrated by evaluating two representative libraries, ‘Extensible and Modular Generics for the Masses’ and SYB Revolutions.

10.1 Future work

A lot of work has been done, however there still some remaining actions. The evaluation of the remaining libraries, as discussed in Section 9.3, needs to be made complete. The results of the evaluations of the libraries could be posted on the generics part of the Haskell website.

The following additional generic programming libraries in Haskell were discovered, or brought to our attention, but haven’t been (fully) analysed and evaluated yet:

- Yet Another Generics Scheme (YAGS) [7],

10 Epilogue

- A pattern for almost compositional functions (Compos) [3].

When all libraries are evaluated, they can be compared to each other. Following the comparison, the next step is to start the design of the common generic programming library.

Bibliography

- [1] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming - An Introduction -. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.
- [2] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Proceedings of the 4th International Conference on Mathematics of Program Construction, MPC'98*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.
- [3] Björn Bringert and Aarne Ranta. A pattern for almost compositional functions. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 216–226, New York, NY, USA, 2006. ACM Press.
- [4] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel Chakravarty, editor, *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.
- [5] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.
- [6] Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löb. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Proceedings Trends in Functional Programming 2006*, 2007.
- [7] Benja Fallenstein. Yet Another Generics Scheme. Posted on the Haskell Café mailinglist, 2007.
- [8] Jeroen Fokker. Functioneel programmeren. Technical report, Utrecht University, 1998.
- [9] Ralf Hinze. Polymorphic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.

BIBLIOGRAPHY

- [10] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The fun of programming*, pages 245–262. Palgrave Macmillan, 2003.
- [11] Ralf Hinze. Generics for the masses. In *Proceedings of the ACM SIG-PLAN International Conference on Functional Programming, ICFP 2004*, pages 236–243. ACM Press, 2004.
- [12] Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–482, 2006.
- [13] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Lecture notes of the Spring School on Datatype-Generic Programming*, LNCS 4719, pages 72–149. Springer-Verlag, 2006.
- [14] Ralf Hinze and Andres Löh. “Scrap Your Boilerplate” revolutions. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC’06*, volume 4014 of *LNCS*, pages 180–208. Springer-Verlag, 2006.
- [15] Ralf Hinze and Andres Löh. Generic programming, now! In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2007.
- [16] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. “Scrap Your Boilerplate” reloaded. In Philip Wadler and Masimi Hagiya, editors, *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS 2006*, volume 3945 of *LNCS*. Springer-Verlag, 2006.
- [17] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, 2000.
- [18] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction, MPC’06*, volume 4014 of *LNCS*, pages 209–234. Springer-Verlag, 2006.
- [19] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. Technical Report UU-CS-2006-020, Utrecht University, 2006.
- [20] Paul Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.

BIBLIOGRAPHY

- [21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM Press, 2007.
- [22] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 470–482. ACM Press, 1997.
- [23] Johan Jeuring and Rinus Plasmeijer. Generic programming for Software Evolution. Technical Report UU-CS-2006-024, Utrecht University, 2006.
- [24] Johan Jeuring, Alexey Rodriguez Yakushev, Oleg Kiselyov, Bruno C. d. S. Oliveira, Patrik Jansson, and Alex Gerdes. Comparing Libraries for Generic Programming. In preparation, 2007.
- [25] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming*, LNCS 1782. Springer-Verlag, 2000.
- [26] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries The Revised Report*. Cambridge University Press, 2003.
- [27] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006.
- [28] Oleg Kiselyov. Smash your boilerplate without class and typeable. <http://article.gmane.org/gmane.comp.lang.haskell.general/14086>.
- [29] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003. TLDI'03.
- [30] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 244–255. ACM Press, 2004.
- [31] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 204–215, 2005.

BIBLIOGRAPHY

- [32] Ralf Lämmel, Eelco Visser, and Joost Visser. The Essence of Strategic Programming. 18 p.; Draft; Available at <http://www.cwi.nl/~ralf>, 2002.
- [33] Ralf Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proceedings of Practical Aspects of Declarative Programming, PADL 2003*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, 2003.
- [34] Ralf Lämmel and Joost Visser. Typed Combinators for Generic Traversal. In *Proceedings Practical Aspects of Declarative Programming, PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, 2002.
- [35] M.M. Lehman and L.A. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985.
- [36] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In Johan Jeuring, editor, *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press.
- [37] The Haskell Generic Library list. Generic programming criteria template. Wiki page at http://www.haskell.org/haskellwiki/Applications_and_libraries/Generic_programming/Template.
- [38] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
- [39] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 141–152, 2003.
- [40] Andres Löh and Ralf Hinze. Open data types and open functions. In Michael Maher, editor, *Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 2006.
- [41] Ian Lynagh. Template Haskell: A report from the field. URL: http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/Template_Haskell-A_Report_From_The_Field.ps, May 2003.
- [42] Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *To be determined*, 2007. Haskell Workshop.
- [43] Pablo Nogueira. Generalised Algebraic Data Types Phantom Types and Dependent Types, January 2007.

BIBLIOGRAPHY

- [44] Ulf Norell and Patrik Jansson. Polytypic programming in Haskell. In Phil Trinder et al., editors, *Implementation of Functional Languages 2003, 15th International Workshop*, volume 3145 of *LNCS*, pages 168–184. Springer-Verlag, 2004.
- [45] Rinus Plasmeijer and Marko van Eekelen. Clean language report version 2.0. Technical report, Department of software technology University of Nijmegen, 2001.
- [46] Alexey Rodriguez Yakushev. A Taste of programming with Generalised Algebraic Datatypes. Technical report, Utrecht University, 2007.
- [47] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.6.1*, 2007. http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html.
- [48] Stephanie Weirich. RepLib: a library for derivable type classes. In Andres Löb, editor, *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12. ACM Press, 2006.
- [49] Noel Winstanley and John Meacham. *DrIFT user guide*, 2006. <http://repetae.net/~john/computer/haskell/DrIFT/>.

Index

Functional programming, 5

INDEX

[2cm]

List of Figures

| | | |
|-----|--|----|
| 2.1 | Hamming numbers implementation | 6 |
| 7.1 | Criteria structure | 60 |

List of Tables

| | | |
|-----|--|----|
| 8.1 | Functions set out against criteria. | 78 |
| 9.1 | Evaluation of generic programming approaches | 89 |