Less Is More

Generic Programming Theory and Practice

José Pedro Magalhães

Promotoren:	Prof.dr. J.T. Jeuring
	Prof.dr. S.D. Swierstra
Co-promotor:	Dr. A. Löh

This work has been supported by the “Fundação para a Ciência e a Tecnologia” (FCT), under grant number SFRH/BD/35999/2007.

Contents

CHAPTER 1

Introduction

A computer is a machine designed to carry out operations automatically. Moreover, a computer is *programmable*, in the sense that it is easy to alter the particular task of a computer. Almost all computers are *digital*, in the sense that they manipulate discrete values, instead of continuous (analogue) ranges. Digital computers evolved significantly in the approximately 70 years since their appearance, and are now an essential part of most people's lives.

At its core, a digital computer can be seen as a relatively simple machine that reads and stores values in memory, while performing arithmetic calculations on these values. The set of instructions to perform is itself also stored in and read from memory. Coupled with suitable input and output devices (such as a keyboard and a monitor), this allows a *programmer* to define a behaviour for the computer to execute. Modern computers can represent data in memory in different formats (such as integers, fractional numbers, or vectors, for instance), and perform various calculations (such as addition, trigonometry functions, and vector multiplication, for instance). Additionally, they can compare values and continue execution depending on the comparison ("if a certain memory value is zero then do this else do that", for instance). A *program* is then a sequence of these instructions.

As advanced as the fundamental operations of modern digital computers might be, they are very far from what most users expect from a computer. Being able to add and subtract numbers is nice, but how do we use this to browse the internet, make a phone call, or to interpret a magnetic resonance exam? Even when computers can perform thousands of millions of primitive operations per second, we certainly do not want to express all tasks in terms of only simple operations. If we want to display some text on the screen, we want to say which characters should show up where; we do not want to have to write specially crafted numbers to a specific location in memory, which is

then interpreted by the screen as a sequence of intensity values for each of the pixels, eventually making up a sequence of characters! Naturally, we need to *abstract* from such details, and be able to formulate programs in a high-level fashion, specifying what should be done, and having the system decide how it should be done.

Abstraction is ubiquitous in computer programming. To allow programmers to abstract from details, several *programming languages* have been developed. Such languages are a precise way to express commands to a computer, and eventually are *translated* into the only primitive operations that the computer understands. The automatic translation process from one programming language to another is the task of a *compiler*, itself a program which reads programs in one input language and outputs programs in another language. Compilers can be chained together, to collectively bring a program from a very high-level programming language (abstract and convenient for human reasoning) down to machine language (low-level and ready to be executed by the computer).

The work of this thesis concerns a special class of very high-level programming languages, namely statically-typed purely functional languages. In such languages, the computation to be performed is described by functions that take input and produce output. The evaluation of a program thus consists of evaluating calls to these functions. This paradigm is rather distant from the machine language at the core of the computer. The machine language deals with sequences of instructions, conditional operations, and loops. A functional language deals with function application, composition, and recursion. We choose functional programming as the starting point for our research because we believe it lends itself perfectly to express abstraction, leading to shorter and more understandable computer programs. This allows the programmer to express complex behaviour in a simple fashion, resulting in programs that are easier to adapt, compose, maintain, and reason about, all desirable properties for computer programs.

We focus on one specific form of abstraction. Computer programs manipulate data, which can either be primitive machine data (such as integer or fractional numbers) or programmer-defined data (such as lists, trees, matrices, images, etc.). There is only a small number of primitive datatypes, but a potentially infinite number of programmer-defined data. The structure of the latter data depends on the problem at hand, and while some structures appear very often (such as sequences of values), others are truly specific to a particular problem.

Some kind of functionality is generally desired for all types of data. Reading and storing files to the disk, for instance, is as important for machine integers as it is for complex healthcare databases, or genealogy trees. And not just reading and writing files: testing for equality, sorting, traversing, computing the length, all are examples of functionality that is often desired for all kinds of data. Most programming languages allow defining complex datatypes as a form of abstraction, but few provide good support for defining behaviour that is *generic* over data. As such, programmers are forced to specify this behaviour over and over again, once for each new type of data, and also to adapt this code whenever the structure of their data changes. This is a tedious task, and can quickly become time-consuming, leading some programmers to write programs to generate this type of functionality automatically from the structure of data.

We think that a programming language should allow programmers to define *generic*

programs, which specify behaviour that is generic over the type of data. Moreover, it should automatically provide generic behaviour for new data, eliminating the need for repeated writing and rewriting of trivial code that just specialises general behaviour to a particular type of data. It should do so in a convenient way for the programmer, leading to more abstract and concise programs, while remaining clear and efficient. This leads us to the two research questions we set out to answer:

1. There are many different approaches to generic programming, varying in complexity and expressiveness. How can we better understand each of the approaches, and the way they relate to each other?

2. Poor runtime efficiency, insufficient datatype support, and lack of proper language integration are often pointed out as deficiencies in generic programming implementations. How can we best address these concerns?

We answer the first question in the first part of this thesis. We start by picking a number of generic programming approaches and define a concise model for each of them. We then use this model to formally express how to embed the structural representation of data of one approach into another, allowing us to better understand the relation between different approaches. The second part of this thesis deals with answering the second question, devoting one chapter to analysing and mitigating each of the practical concerns.

1.1 Choice of programming language

While functional languages are not the most popular for program development, they are ahead of popular languages in terms of the abstraction mechanisms they provide. We have also observed that popular languages, when adding new features, often borrow features first introduced and experimented with in functional languages. While we do not believe that the developments in this thesis will directly lead to widespread adoption of generic programming, we hope that they serve to further advance the field and provide further evidence that abstraction over the shape of data is an essential aspect of modern software development.

All the code in this thesis is either in Haskell [Peyton Jones, 2003], a statically-typed purely functional language, or Agda [Norell, 2007], a dependently-typed purely functional language. We use Agda in the first part of this thesis, since the first part deals with modelling typed representations, and many concepts that arise at the type level are easy to express in Agda because it is a dependently-typed language. Additionally, due to Agda's foundation in constructive type theory and the Curry-Howard isomorphism, we can use it for expressing formal proofs for some of our code.

We use Haskell as the implementation language for the practical aspects of this thesis, in the dialect implemented by the Glasgow Haskell Compiler (GHC).[1] Haskell is a fully

[1] `http://www.haskell.org/ghc/`

developed language with serious industrial applications, offering many advanced type-level programming features required for generic programming that are absent in most other languages. Haskell has a vibrant community of both users and researchers, which provides continuous feedback for what is relevant, necessary, and desired from generic programming in the wild.

This thesis is written in literate programming style using `lhs2TeX`.[2] The resulting code files and other additional material can be found online at `http://dreixel.net/thesis`.

1.2 Relation with previously published material

Some parts of this thesis are based on a number of previously refereed and published research articles. The author of this thesis is a main co-author in all of those articles. The first part of the thesis follows the structure of Magalhães and Löh [2012], with Chapter 8 being at its core. Chapter 6 is based on Löh and Magalhães [2011]. Chapter 9 is based on Magalhães et al. [2010b], but it has been completely rewritten to focus on a single approach and to avoid using compilation flags that are pervasive through the whole program. Chapter 10 is a revised version of Magalhães and Jeuring [2011], where a section originally discussing the use of unsafe casts has been replaced by a discussion on how to support datatypes with indices of custom kinds. Chapter 11 is an updated version of Magalhães et al. [2010a], reflecting the current support for generic programming in two Haskell compilers.

1.3 Roadmap

This thesis is divided in two parts, each part answering one research question.

We start the first part with a gentle introduction to generic programming (Chapter 2), also introducing Agda along the way. We then present five generic programming approaches: `regular` (Chapter 3), `polyp` (Chapter 4), `multirec` (Chapter 5), `indexed` (Chapter 6), and `instant-generics` (Chapter 7). We present each approach by first showing how it can be modelled in Agda, and then give its Haskell encoding. The description of `indexed` lacks a Haskell encoding because Haskell is not (yet) capable of faithfully encoding this approach. On the other hand, we give more examples of generic functionality for `indexed`, namely a zipper for efficient navigation of data structures, and an algorithm for decidable equality. Finally, Chapter 8 presents a formal relation between the five approaches, showing, for instance, that the functionality of `indexed` can be used in other approaches.

In the second part we focus on three main issues that often prevent a wider adoption of generic programming. Chapter 9 explores the potential for optimisation of generic programs, with the aim of removing all runtime overhead of genericity through optimisation at compile time. Chapter 10 deals with extending datatype support for a representative generic programming library, adding the ability to work generically with indexed

[2] `http://www.andres-loeh.de/lhs2tex/`

datatypes. Finally, Chapter 11 presents a way to embed generic programming more seamlessly within Haskell, simplifying generic function definition and usage from the user's point of view. We conclude in Chapter 12 with a few remarks on the future of generic programming.

Part I

Generic programming libraries

CHAPTER 2

Introduction to generic programming

In this chapter we introduce the concept of *generic programming* through a series of examples in Agda, a dependently typed functional programming language.

Colours and code highlighting

Throughout this thesis we use this font (Helvetica) for code blocks. Keywords are highlighted in **bold**, and we use three different colours in code blocks to distinguish identifiers.

Haskell code

In Haskell, there is a clear distinction between values, types, and kinds. Values are built with constructors, such as True, which we colour in blue; we do not colour numerals, characters, or strings: Just 3, (Right 'p' , Left "abc").

Datatypes, type synonyms, type variables, indexed families, and type classes are coloured in orange: Bool, Show $\alpha \Rightarrow$ Show [α], **type family** PF α.

Kinds and kind variables are coloured in green: $\star$, $\star \to \star$, Constraint. For simplicity we do not colour the arrow operator, as it can appear at different levels.

Agda code

The distinction between values and types is less clear in Agda. Our convention is to use blue for constructors (e.g. refl), orange for identifiers of type Set (e.g. _≡_), and green for identifiers of type Set_1 or (for simplicity) higher (e.g. Set).

2.1 What is generic programming?

The term "generic programming" has been used for several different but related concepts. Gibbons [2007] provides a detailed account which we summarise here. In general, all forms of generic programming deal with increasing program abstraction by generalising, or parametrising, over a concept.

Value genericity Arguably the simplest form of generalisation is parametrisation by value. Consider first the type of natural numbers defined by induction:

```
data ℕ : Set where
  ze : ℕ
  su : ℕ → ℕ
```

This declaration defines a new type ℕ that is inhabited by the ze element, and an operation su that transforms one natural number into another. We call ze and su the constructors of the type ℕ, as they are the only way we can build inhabitants of the type.

Consider now an operator that computes the sum of two natural numbers:

```
_+_ : ℕ → ℕ → ℕ
m + ze     = m
m + (su n) = su (m + n)
```

The _+_ operator (with underscores denoting the position of its expected arguments) is our first example of generalisation, as it defines an operation that works for any two natural numbers. Value abstraction is generally not called "generic programming", as it is nearly universally present in programming languages, and as such is not considered special in any way. It serves however as a very basic example of the concept of abstraction: one that is crucial in programming languages in general, and in generic programming in particular.

Type genericity Suppose now we are interested in handling sequences of natural numbers. We can define such a type inductively in Agda as follows:

```
data ℕList : Set where
  nil  : ℕList
  cons : ℕ → ℕList → ℕList
```

Now we can define lists of natural numbers. For example, nil encodes the empty list, and cons 0 (cons 1 (cons 2 nil)) encodes the list of natural numbers from zero to two. Note that for convenience we are using numerals such as 0 and 1 instead of ze and su ze.

With ℕList we are limited to encoding sequences of natural numbers, but the concept of a list is not in any way specific to the naturals. Since Agda has a polymorphic type system, we can also define a "type generic" list datatype:

```
data [_] (α : Set) : Set where
  []  : [ α ]
  _::_ : α → [ α ] → [ α ]
```

This parametric [_] type encodes lists of any Set α; in particular, [ℕ] is equivalent (in the formal sense of Section 2.2.3) to the ℕList type we have just defined. We use the symbol [] to represent the empty list, while _::_ is the operator that adds an element to a list.

This form of parametrisation is often called simply "generics" in languages such as Java [Bracha et al., 2001] and C# [Kennedy and Syme, 2001], but it is more commonly referred to as "parametric polymorphism" in Haskell or Agda.

Functions operating on polymorphic datatypes can also be polymorphic. For instance, a function that computes the length of a list does not care about the particular type of elements in the list:

```
length : {α : Set} → [ α ] → ℕ
length []       = 0
length (h :: t) = 1 + length t
```

Functions such as length are called polymorphic functions. In Agda, type arguments to functions have to be explicitly introduced in the type declarations. So we see that length takes a Set α, a list of αs, and returns a natural number. The curly braces around α indicate that it is an implicit argument, and as such it does not appear as an argument to length. The Agda compiler tries to infer the right implicit arguments from the context; when it fails to do so, the programmer can supply the implicit arguments directly by wrapping them in curly braces.

Gibbons distinguishes also "function genericity", where the polymorphic arguments of functions can themselves be functions. A simple example is the function that applies a transformation to all elements of a list:

```
map : {α β : Set} → (α → β) → [ α ] → [ β ]
map f []       = []
map f (h :: t) = f h :: map f t
```

This function transforms a list containing αs into a list of the same length containing βs, by applying the parameter function f to each of the elements. The map function is akin to the loop operation in imperative programming languages, and can be used for many distinct operations. For example, to increment each of the naturals in a list we can use the map su function, and if we have a function toChar : ℕ → Char that converts natural numbers to characters, we can transform a list of naturals into a string by applying the function map toChar. These are simple examples, but they already reveal the power of abstraction and generalisation in reducing code duplication while increasing code clarity.

Structure genericity When we pair polymorphism with abstract specifications we get to the so-called "C++ generics". The Standard Template Library [Austern, 1999] is the

primary example of structure genericity in C++, which consists of specifying abstract data structures together with standard traversal patterns over these structures, such as iterators. The abstract structures are containers such as lists, vectors, sets, and maps, whereas the iterators provide support for accessing the contained elements in a uniform way. Unlike our [_] datatype, these abstract containers do not specify constructors; the only way to operate on them is through the iterators.

This form of structure genericity is commonly called "generic programming" within the C++ community [Siek et al., 2002], and is also known as "overloading" in other languages. In Haskell, a similar programming style can be achieved through the use of type classes, and in Agda through the use of module parameters. This is, however, not the style of generic programming that is the subject of this thesis. Although structure genericity allows for easy switching between different types of containers, it does not necessarily reduce code duplication, for instance, as similar containers might still have to separately implement similar operations, without any possibility of code reuse.

Stage genericity A metaprogram is a program that writes or manipulates other programs. In combination with some form of staging or splicing, this allows for programs that manipulate themselves, or perform computations during type checking. Template Haskell [Sheard and Peyton Jones, 2002] is an implementation of metaprogramming in GHC; Agda has a reflection mechanism with similar expressiveness.[1]

Metaprogramming mechanisms can be very expressive and allow for defining generic programs in all the senses of this section. For instance, one can define a Template Haskell program that takes a container type and then defines an appropriate length function for that type. This could in principle be done even in a non-polymorphic language. However, encoding generic behavior through metaprogramming is a tedious and error-prone task, since it involves direct manipulation of large abstract syntax trees.

Shape genericity The type of genericity we focus on arises from abstracting over the *shape* of datatypes. Consider a datatype encoding binary leaf trees:

```
data Tree (α : Set) : Set where
  leaf : α → Tree α
  bin  : Tree α → Tree α → Tree α
```

For the list datatype we have seen length and map functions. However, these functions are not specifically exclusive to lists; we can equally well define them for trees:

```
length : {α : Set} → (Tree α) → ℕ
length (leaf x) = 1
length (bin l r) = length l + length r

map : {α β : Set} → (α → β) → (Tree α) → (Tree β)
map f (leaf x) = leaf (f x)
map f (bin l r) = bin (map f l) (map f r)
```

[1] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Reflection`

In fact, length and map are only two of many functions that are generic over the structure of datatypes. Note how the structure of the functions is determined by the datatype itself: for length, we return 0 for constructors without arguments, 1 at any occurrence of the parameter, and call length recursively at the recursive positions of the datatype in question, summing the results obtained for all the components of each constructor. For map, we keep the structure of the datatype, applying the mapping function to the parameters and recursively calling map at recursive positions.

This is the type of generic programming we are interested in: programs that generalise over the shape of datatypes. This style of programming is also called "datatype-generic", "polytypic", and "type-indexed" programming [Dos Reis and Järvi, 2005]. From this point on, when we refer to "generic programming" we mean this form of genericity.

So far we have only shown two definitions of both map and length, and argued that they really should be a single function each. We now proceed to show precisely how that can be done.

2.2 Categorical background

Generic programming arose from the categorical concepts of F-algebras and homomorphisms. A detailed introduction to category theory and how it relates to functional programming in general and datatypes in particular is outside the scope this thesis; the reader is referred to Backhouse et al. [1999], Bird and Moor [1997], Fokkinga [1992], and Meertens [1995] for a precise account of these foundations. Here we focus on a practical encoding of functors for constructing datatypes, up to simple recursive morphisms and some of their laws.

2.2.1 Polynomial functors

The key aspect of generic programming is in realising that all datatypes can be expressed in terms of a small number of primitive type operations. In order to handle recursion, as we will see later, the primitive types are all functors:

```
Functor : Set₁
Functor = Set → Set
```

For the purposes of this section, we define a Functor to be a function on sets.[2] The Functor type is not a Set since it contains sets itself, therefore it must be in Set_1 (see Chlipala [2012] for more information on hierarchical universes). Consider now the following encoding of a lifted sum type:

```
data _⊕_ (φ ψ : Functor) (ρ : Set) : Set where
  inj₁ : φ ρ → (φ ⊕ ψ) ρ
  inj₂ : ψ ρ → (φ ⊕ ψ) ρ
```

[2] Categorically speaking, this is the functor operation on objects. On the other hand, the Haskell Functor class represents the categorical functor operation on morphisms. Normally functors also have associated laws, which we elide for the purposes of this section, but revisit in Section 6.2.

This is a "lifted" type because it deals with Functors, instead of just sets. Its arguments ϕ and ψ are also Functors, and the sum $\phi \oplus \psi$ is itself of type Functor. It is called a "sum" because it shares many properties with the usual algebraic sum; another common name is "disjoint union". Its dual is the lifted product:

```
data _⊗_ (φ ψ : Functor) (ρ : Set) : Set where
  _,_ : φ ρ → ψ ρ → (φ ⊗ ψ) ρ
```

Again, the name "product" refers to its similarity to the algebraic product, or the set product. Finally, consider the encoding of a lifted unit type:

```
data 𝟙 (ρ : Set) : Set where
  1 : 𝟙 ρ
```

We call this a "unit" type because it has a single inhabitant, namely the 1 constructor. It is also lifted, as evidenced by its argument ρ.

These basic three datatypes, which we call "representation" functors, can be used to encode many others. Consider a very simple datatype for encoding Boolean values:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

To encode a Boolean we have a choice between two possible values: true and false. If we ignore the constructor names, this is equivalent to the sum of two units:

```
Bool_F : Functor
Bool_F = 𝟙 ⊕ 𝟙
```

The type $\mathsf{Bool}_F\ \rho$ also has only two inhabitants, namely $\mathsf{inj}_1\ 1$ and $\mathsf{inj}_2\ 1$. The algebraic nature of our representation functors also becomes evident: the type $\mathbb{1}$ has one inhabitant, and the type $\mathbb{1} \oplus \mathbb{1}$ has two inhabitants.

2.2.2 Recursion

The $\mathbb{N}$ datatype of Section 2.1 is a choice between ze or su. We can also encode this datatype as a sum, but we are left with the problem of dealing with recursion. Basically, we want to encode the following recursive equation:

$$\mathsf{Nat}_F = \mathbb{1} \oplus \mathsf{Nat}_F$$

For this we first need an identity functor:

```
data Id (ρ : Set) : Set where
  id : ρ → Id ρ
```

This datatype, which simply stores the parameter ρ, will be used in place of the recursive occurrences of the datatype being coded. For $\mathbb{N}$ we thus get:

```
Natᶠ : Functor
Natᶠ = 𝟙 ⊕ Id
```

The functorial nature of our representation types is now clear. The type Nat_F is not recursive; however, with the right instantiation of the parameter ρ, we can define $\mathbb{N}$ in terms of Nat_F. For this we need a fixed-point operator:

```
data μ (ϕ : Functor) : Set where
  ⟨_⟩ : ϕ (μ ϕ) → μ ϕ

out : {ϕ : Functor} → μ ϕ → ϕ (μ ϕ)
out ⟨ x ⟩ = x
```

This primitive recursive operator, with constructor $\langle \_ \rangle$ and destructor out, can be used to tie the recursive knot of our representation type Nat_F. In fact, $\mu\ \mathsf{Nat}_F$ is equivalent to $\mathbb{N}$, as it encodes the infinite expansion $\mathbb{1} \oplus \mathbb{1} \oplus \mathbb{1} \oplus \ldots$, which we obtain by replacing each occurrence of Id in Nat_F with the expansion of Nat_F itself.

The type $[\,\_\,]$ of lists can be encoded similarly, but first we need one more representation type for constants:

```
data K (α ρ : Set) : Set where
  k : α → K α ρ
```

The type K is a lifted constant operator, which we use for encoding the type parameter of lists. The encoding of lists $[\,\_\,]_F$ uses all the representation types we have seen:

```
[ _ ]ᶠ : Set → Functor
[ α ]ᶠ = 𝟙 ⊕ (K α ⊗ Id)
```

The [] constructor is encoded as a unit, hence the $\mathbb{1}$, whereas the _::_ constructor has two arguments, so we use a product. The first argument is a parameter of type α, which we encode as a constant, and the second argument is again a list, so we use Id. The fixed point $\mu\ [\,\alpha\,]_F$ corresponds to the algebraic expansion $\mathbb{1} \oplus (\mathsf{K}\ \alpha \otimes (\mathbb{1} \oplus (\mathsf{K}\ \alpha \otimes \ldots)))$, which is equivalent to $[\,\alpha\,]$.

2.2.3 Isomorphisms

We have mentioned that some types are *equivalent* to others without formally defining this equivalence. When we say two types are equivalent we mean that they are *isomorphic*. The concept of isomorphism can be succinctly summarised as an Agda record type:

```
record _≃_ (α β : Set) : Set where
  field
    from : α → β
    to   : β → α
```

```
iso₁ : ∀ x → to (from x) ≡ x
iso₂ : ∀ x → from (to x) ≡ x
```

An isomorphism $\alpha \simeq \beta$ is a tuple of four components: two functions that convert between α and β, and two proofs iso_1 and iso_2 that certify the cancellation of the composition of the conversion functions. We call iso_1 and iso_2 proofs because their type is a propositional equality; the _≡_ type is defined as follows:

```
data _≡_ {α : Set} (x : α) : α → Set where
  refl : x ≡ x
```

The only way to build a proof of propositional equality refl is when the arguments to _≡_ are equal. In this way, if we can give a definition for iso_1, for instance, we have proven that to (from x) equals x.

Previously we have stated that $\mathbb{N}$ and $\mu\ \mathsf{Nat}_\mathsf{F}$ are equivalent. We can now show that they are indeed isomorphic. We start by defining the conversion functions:

```
fromℕ : ℕ → μ Natꟳ
fromℕ ze     = ⟨ inj₁ 1 ⟩
fromℕ (su n) = ⟨ inj₂ (id (fromℕ n)) ⟩
toℕ : μ Natꟳ → ℕ
toℕ ⟨ inj₁ 1 ⟩       = ze
toℕ ⟨ inj₂ (id n) ⟩ = su (toℕ n)
```

The conversion from $\mathbb{N}$ starts by applying the fixed-point constructor ⟨_⟩, then injecting each constructor in the respective component of the sum, and then stops with a 1 for the ze case, or proceeds recursively through id for the su case. The conversion to $\mathbb{N}$ is trivially symmetrical.

We are left with proving that these conversions are correct. For this we have to build a proof, or an element of the _≡_ type. We already know one such element, refl, which can be used when Agda can directly determine the structural equality of the two terms. However, most often the arguments are not immediately structurally equivalent; then we need to convince the compiler of their equality by building an explicit proof derivation. For this we can use combinators like the following:

```
sym : {α β : Set} → α ≡ β → β ≡ α
sym refl = refl
trans : {α β γ : Set} → α ≡ β → β ≡ γ → α ≡ γ
trans refl refl = refl
cong : {α β : Set} {x y : α} → (f : α → β) → x ≡ y → f x ≡ f y
cong f refl = refl
```

The first two combinators, sym and trans, encode the usual relational concepts of symmetry and transitivity. The cong combinator (short for congruence) is generally used to move the focus of the proof over a Functor. We use it in the isomorphism proof for naturals:

```
isoℕ₁ : (n : ℕ) → toℕ (fromℕ n) ≡ n
isoℕ₁ ze     = refl
isoℕ₁ (su n) = cong su (isoℕ₁ n)
isoℕ₂ : (n : μ Natꟳ) → fromℕ (toℕ n) ≡ n
isoℕ₂ ⟨ inj₁ 1 ⟩      = refl
isoℕ₂ ⟨ inj₂ (id n) ⟩ = cong (λ x → ⟨ inj₂ (id x) ⟩) (isoℕ₂ n)
```

For the ze case we can directly use refl, but for the su case we have to proceed recursively, using cong to move the focus of the proof over the functor to the recursive position.

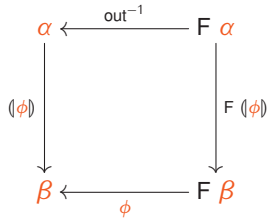
We now have all the components necessary to form an isomorphism between $\mathbb{N}$ and $\mu\ \mathsf{Nat_F}$:

```
isoℕ : ℕ ≃ μ Natꟳ
isoℕ = record {from = fromℕ; to  = toℕ
              ; iso₁ = isoℕ₁ ; iso₂ = isoℕ₂}
```

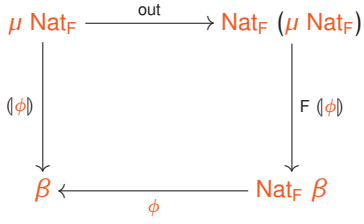
The isomorphism between $[\ \alpha\ ]$ and $\mu\ [\ \alpha\ ]_\mathsf{F}$ can be constructed in a similar way.

2.2.4 Recursive morphisms

Many generic functions are instances of a *recursive morphism* [Meijer et al., 1991]. These are standard ways of performing a recursive computation over a datatype. We start with *catamorphisms*, based on the theory of initial F-algebras. Categorically speaking, the existence of an initial F-algebra $\mathsf{out}^{-1} : \mathsf{F}\ \alpha \to \alpha$ means that for any other F-algebra $\phi : \mathsf{F}\ \beta \to \beta$ there is a unique homomorphism from out^{-1} to ϕ. This information can be summarised in a commutative diagram:



If we invert the direction of out^{-1} and instantiate it to the type $\mu\ \mathsf{Nat_F} \to \mathsf{Nat_F}\ (\mu\ \mathsf{Nat_F})$ we get the definition of catamorphism for natural numbers:



To be able to encode this diagram as a function, we are still lacking a functorial mapping function, used in the arrow on the right of the diagram. For the natural numbers, we are looking for a function with the following type:

```
mapℕ : {α β : Set} → (α → β) → Nat_F α → Nat_F β
```

This function witnesses the fact that Nat_F is a functor in the categorical sense, mapping a morphism between Sets to a morphism between Nat_Fs. It is also a generic function, in the sense that its definition is determined by the structure of the datatype it applies to. Since Nat_F is $\mathbb{1} \oplus \mathsf{Id}$, to define mapℕ we first need to define map on units, sums, and identity:

```
map_𝟙 : {ρ ρ′ : Set} → (ρ → ρ′) → 𝟙 ρ → 𝟙 ρ′
map_𝟙 _ 1 = 1

map_⊕ : {ϕ ψ : Functor} {ρ ρ′ : Set} →
        (ϕ ρ → ϕ ρ′) → (ψ ρ → ψ ρ′) → (ϕ ⊕ ψ) ρ → (ϕ ⊕ ψ) ρ′
map_⊕ f g (inj₁ x) = inj₁ (f x)
map_⊕ f g (inj₂ x) = inj₂ (g x)

map_Id : {ρ ρ′ : Set} → (ρ → ρ′) → Id ρ → Id ρ′
map_Id f (id x) = id (f x)
```

Units do not contain any ρ, so we do not map anything. We choose to define $\mathsf{map}_{\mathbb{1}}$ taking the function argument for consistency only; it is otherwise unnecessary. The fact that we do not use it in the definition is also made clear by pattern-matching on an underscore. For sums we need to know how to map on the left and on the right. For an identity we simply apply the function.

We can now define mapℕ generically:

```
mapℕ : {α β : Set} → (α → β) → Nat_F α → Nat_F β
mapℕ f = map_⊕ (map_𝟙 f) (map_Id f)
```

In the same style as for Nat_F, we also have a mapping function for $[\,\alpha\,]_F$, or any other functor written using the polynomial functors we have introduced so far, because these (aptly named) polynomial functors are functorial. The definition of catamorphism for $\mathbb{N}$ follows:

```
cataℕ : {β : Set} → (Nat_F β → β) → μ Nat_F → β
cataℕ ϕ = ϕ ∘ mapℕ (cataℕ ϕ) ∘ out
```

Note how the definition directly follows the diagram, using function composition (represented in Agda by the operator _∘_) for composing the three arrows on the right.[3]

[3] The definition of cataℕ does not pass Agda's termination checker. For practical purposes we disable the termination checker and do not guarantee termination for our functions. Section 8.4 presents a small discussion on the implications of this decision on our proofs.

In this chapter we have defined the term "generic programming" and seen some concrete examples. However, we have manually defined structural representation types, and manually instantiated generic functions. In the next chapter we show a practical encoding of these same concepts in a real library, including the necessary infrastructure to derive generic definitions of map and catamorphism automatically.

CHAPTER 3

The `regular` library

In this chapter we introduce the `regular` library for generic programming. We start by defining a concise model of the library's core in Agda, and then show practical aspects of its implementation in Haskell. This is the general structure of each of the five chapters introducing generic programming libraries in this thesis: we first introduce an Agda model of the library, and then show the details of the Haskell implementation.[1] The Agda model focuses on core concepts such as the generic view [Holdermans et al., 2006] used and the datatype support offered. We show it first as it serves also as a theoretical overview of the library at hand. The Haskell encoding deals with more practical aspects such as ease of use and performance.

3.1 Agda model

`regular` is a simple generic programming library, originally developed to support a generic rewriting system [Van Noort et al., 2008]. It has a fixed-point view on data: the generic representation is a pattern-functor, and a fixed-point operator ties the recursion explicitly. In the original formulation, this is used to ensure that rewriting meta-variables can only occur at recursive positions of the datatype. This fixed-point view on data is very close to the categorical view on datatypes introduced in the previous chapter. The library name derives from the concept of a regular datatype, which is a type that can be represented as a potentially recursive polynomial functor. This definition excludes exponentials (functions), or nested datatypes [Bird and Meertens, 1998], amongst others, which `regular` indeed does not support.

[1] Chapter 6 follows a slightly different structure; the reason for the different treatment is explained there.

We model each library by defining a Code type that represents the generic universe, and an interpretation function ⟦_⟧ that maps codes to Agda types. The universe and interpretation for `regular` follow, side by side:

```
data Code : Set where
  U   : Code
  I   : Code
  _⊕_ : (F G : Code) → Code
  _⊗_ : (F G : Code) → Code
```

```
⟦_⟧ : Code → (Set → Set)
⟦ U     ⟧ α = ⊤
⟦ I     ⟧ α = α
⟦ F ⊕ G ⟧ α = ⟦ F ⟧ α ⊎ ⟦ G ⟧ α
⟦ F ⊗ G ⟧ α = ⟦ F ⟧ α × ⟦ G ⟧ α
```

We have codes for units, identity (used to represent the recursive positions), sums, and products. The interpretation of unit, sum, and product relies on the Agda types for unit (⊤), disjoint sum (_⊎_), and non-dependent product (_×_), respectively. The interpretation is parametrised over a Set α that is returned in the I case. We omit a case for constants for simplicity of presentation only.

As before, we need a fixed-point operator to tie the recursive knot. We define a μ operator specialised to `regular`, taking a Code as argument and computing the interpretation of its least fixed point:

```
data μ (F : Code) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

In Section 2.2.4 we have introduced the recursive morphisms, and how they can be defined from a map function. In `regular`, this function lifts a mapping between sets α and β to a mapping between interpretations parametrised over α and β, simply by applying the function in the I case:

```
map : (F : Code) → {α β : Set} → (α → β) → ⟦ F ⟧ α → ⟦ F ⟧ β
map U       f _       = tt
map I       f x       = f x
map (F ⊕ G) f (inj₁ x) = inj₁ (map F f x)
map (F ⊕ G) f (inj₂ y) = inj₂ (map G f y)
map (F ⊗ G) f (x , y)  = map F f x , map G f y
```

Note that tt is the only constructor of the ⊤ type, inj_1 and inj_2 are the constructors of _⊎_, and _,_ is the constructor of _×_.

We can now give a truly generic definition of catamorphism:

```
cata : {α : Set} → (F : Code) → (⟦ F ⟧ α → α) → μ F → α
cata F f ⟨ x ⟩ = f (map F (cata F f) x)
```

This definition of cata works for any `regular` Code, using the generic definition of map.

Datatypes can be encoded by giving their code, such as NatC for the natural numbers, and then taking the fixed point:

```
NatC : Code
NatC = U ⊕ I
```

Hence, a natural number is a value of type μ NatC; in the example below, aNat encodes the number 2:

```
aNat : μ NatC
aNat = ⟨ inj₂ ⟨ inj₂ ⟨ inj₁ tt ⟩ ⟩ ⟩
```

3.2 Haskell implementation

In Haskell, the universe and interpretation of `regular` is expressed by separate datatypes:

```
data U         α = U
data I         α = I α
data K β       α = K β
data (ϕ :+: ψ) α = L (ϕ α) | R (ψ α)
data (ϕ :×: ψ) α = ϕ α :×: ψ α
```

These definitions are not too different from the code and interpretation in the Agda model. The left-hand side of the datatype declarations are the codes, whereas the right-hand side is the interpretation. We add a K code for embedding arbitrary constant types in the universe; this is essential in a practical implementation to handle primitive types (such as Int or Char).

Note that while we use only these representation types to represent `regular` codes, nothing prevents us from writing types such as (Maybe :+: U) Int, for instance, even though this does not make sense since Maybe is not a representation type. In the Agda model these mistakes are impossible because Maybe is not of type Code and attempting to use it as such would trigger a type error. Note also that since the original implementation of the `regular` library in Haskell there have been new developments that do allow preventing this type of errors [Yorgey et al., 2012], but these have not yet been used to improve `regular`.

The library also includes two representation types for encoding meta data, such as constructor names and fixities, that are necessary for defining generic functions such as show and read. For conciseness we do not make use of those in our description; in Section 11.2.2 we describe in detail how to handle meta-data information within representation types.

3.2.1 Generic functions

Generic functions are written by induction over the universe types. To give instances for the different types we use a type class. For instance, the map function for `regular`

corresponds directly to the standard Haskell Functor class:[2]

```
class Functor φ where
  fmap :: (α → β) → φ α → φ β
```

The implementation of the fmap function in `regular` is given by defining an instance for each of the representation types:

```
instance Functor U where
  fmap _ U       = U
instance Functor I where
  fmap f (I r)   = I (f r)
instance Functor (K α) where
  fmap _ (K x)   = K x
instance (Functor φ, Functor ψ) ⇒ Functor (φ :+: ψ) where
  fmap f (L x)   = L (fmap f x)
  fmap f (R x)   = R (fmap f x)
instance (Functor φ, Functor ψ) ⇒ Functor (φ :×: ψ) where
  fmap f (x :×: y) = fmap f x :×: fmap f y
```

The instances are very similar to the map function in our Agda model, albeit more verbose due to the need of writing the instance heads.

3.2.2 Datatype representation

To convert between datatypes and their generic representation we use another type class. Since `regular` encodes the regular types, we call this class Regular too:

```
class Regular α where
  type PF α :: ⋆ → ⋆
  from      :: α       → PF α α
  to        :: PF α α  → α
```

The class has an associated datatype [Schrijvers et al., 2008] PF, which stands for *pattern functor*, and is the representation of the type in terms of our universe codes. `regular` adopts a shallow encoding, so we have a one-layer generic structure containing user datatypes at the leaves. This is clear in the type of from, for instance: it transforms a user datatype α into a pattern functor (PF α) α, which is the generic representation of α containing α types in the I positions.

As an example, consider the definition and generic representation of natural numbers:

[2] Functor instances in Haskell are often used to allow mapping a function to the elements contained within a type constructor. On the other hand, the Functor instances for `regular` types map over the recursive positions of a datatype.

```
data ℕ = Ze | Su ℕ

instance Regular ℕ where
  type PF ℕ = U :+: I
  from Ze     = L U
  from (Su n) = R (I n)
  to (L U)    = Ze
  to (R (I n)) = Su n
```

The pattern functor of ℕ is a choice between unit (for Ze) and a recursive position (for Su). The conversion functions are mostly unsurprising, but note that in R (I n), the n is of type ℕ, and not of type PF ℕ ℕ. Note also that the conversion functions are not recursive; to fully convert a value to its generic representation we need to map from recursively:

```
data μ φ = In (φ (μ φ))
deepFromℕ :: ℕ → μ (PF ℕ)
deepFromℕ = In ∘ fmap deepFromℕ ∘ from
```

With a fixed-point operator μ we can express the type of a fully converted value, namely μ (PF ℕ). In practice, however, `regular` does not make use of deep conversion functions.

3.2.3 Generic length

In Section 2.1, we promised to show generic definitions for functions map and length that would work on multiple types. We have already shown map, and we can show length too:

```
class LengthAlg φ where
  lengthAlg :: φ Int → Int

instance LengthAlg U where
  lengthAlg _ = 0
instance LengthAlg I where
  lengthAlg (I n) = n + 1
instance LengthAlg (K α) where
  lengthAlg _ = 0
instance (LengthAlg φ, LengthAlg ψ) ⇒ LengthAlg (φ :+: ψ) where
  lengthAlg (L x) = lengthAlg x
  lengthAlg (R x) = lengthAlg x
instance (LengthAlg φ, LengthAlg ψ) ⇒ LengthAlg (φ :×: ψ) where
  lengthAlg (x :×: y) = lengthAlg x + lengthAlg y
```

The lengthAlg function operates on generic representations and computes their length. We assume we already know the length of each recursive position, and add up the length

of multiple arguments. However, lengthAlg is only the algebra of the computation. To apply it recursively over a term we use the catamorphism, this time in Haskell:

```
cata :: (Regular α, Functor (PF α)) ⇒ (PF α β → β) → α → β
cata f x = f (fmap (cata f) (from x))
```

We now have all the elements in place to define generic length for every Regular datatype:

```
length :: (Regular α, Functor (PF α), LengthAlg (PF α)) ⇒ α → Int
length = cata lengthAlg
```

We have only seen one Regular instance, namely for $\mathbb{N}$; as an example, length Su (Su (Su Ze)) evaluates to 3. Consider now a Regular instance for lists:

```
instance Regular [ α ] where
  type PF [ α ] = U :+: (K α :×: I)
  from []       = L U
  from (h : t)  = R (K h :×: I t)
  to (L U)              = []
  to (R (K h :×: I t))  = h : t
```

With this Regular instance for lists we can compute the length of lists, generically. For instance, length [()] evaluates to 1, and length [0, 2, 4] to 3.

We have revisited the foundations of generic programming from Chapter 2 and cast them in the setting of a concrete, simple generic programming library, supporting only regular datatypes. In the following chapters we will present more complex approaches to generic programming, with increased flexibility and datatype support.

CHAPTER 4

The polyp approach

polyp [Jansson and Jeuring, 1997] is an early pre-processor approach to generic programming. As a pre-processor, it takes as input a file containing Haskell code and some special annotations, and outputs a Haskell file, ready to be handled by a full interpreter or compiler. Haskell evolved since 1997, and it is now possible to encode polyp as a library in a similar fashion to regular. In fact, polyp has a very similar generic view to that of regular, only that it abstracts also over one datatype parameter, in addition to one recursive position. polyp therefore represents types as bifunctors, whereas regular uses plain functors.

4.1 Agda model

The encoding of polyp's universe in Agda follows:

```
data Code : Set where
  U    : Code
  P    : Code
  I    : Code
  _⊕_  : (F G : Code) → Code
  _⊗_  : (F G : Code) → Code
  _◎_  : (F G : Code) → Code
```

```
⟦_⟧ : Code → (Set → Set → Set)
⟦ U     ⟧ α β = ⊤
⟦ P     ⟧ α β = α
⟦ I     ⟧ α β = β
⟦ F ⊕ G ⟧ α β = ⟦ F ⟧ α β ⊎ ⟦ G ⟧ α β
⟦ F ⊗ G ⟧ α β = ⟦ F ⟧ α β × ⟦ G ⟧ α β
⟦ F ◎ G ⟧ α β = μ F (⟦ G ⟧ α β)
```

In the codes, the only differences from regular are the addition of a P code, for the parameter, and a code _◎_ for composition. The interpretation is parametrised over two Sets, one for the parameter and the other for the recursive position. Composition is

interpreted by taking the fixed-point of the left bifunctor, thereby closing its recursion, and replacing its parameters by the interpretation of the right bifunctor. The fixed-point operator for polyp follows:

```
data μ (F : Code) (α : Set) : Set where
  ⟨_⟩ : ⟦ F ⟧ α (μ F α) → μ F α
```

There is at least one other plausible interpretation for composition, namely interpreting the left bifunctor with closed right bifunctors as parameter (⟦ F ⟧ (μ G α) β), but we give the interpretation taken by the original implementation.

This asymmetric treatment of the parameters to composition is worth a detailed discussion.[1] In polyp, the left functor F is first closed under recursion, and its parameter is set to be the interpretation of the right functor G. The parameter α is used in the interpretation of G, as is the recursive position β. Care must be taken when using composition to keep in mind the way it is interpreted. For instance, if we have a code for binary trees with elements at the leaves TreeC, and a code for lists ListC, one might naively think that the code for trees with lists at the leaves is TreeC ⊙ ListC, but that is not the case. Instead, the code we are after is (ListC ⊙ P) ⊕ (I ⊗ I). Although the resulting code quite resembles that of trees, this composition does not allow us to reuse the code for trees when defining trees with lists. The indexed approach (Chapter 6) has a more convenient interpretation of composition; this subtle difference is revealed explicitly in our comparison between polyp and indexed (Section 8.2.3).

The fixed-point operator takes a bifunctor and produces a functor, by closing the recursive positions and leaving the parameter open. The map operation for bifunctors takes two argument functions, one to apply to parameters, and the other to apply to recursive positions:

```
map : {α β γ δ : Set}
      (F : Code) → (α → β) → (γ → δ) → ⟦ F ⟧ α γ → ⟦ F ⟧ β δ
map U       f g tt       = tt
map P       f g x        = f x
map I       f g x        = g x
map (F ⊕ G) f g (inj₁ x) = inj₁ (map F f g x)
map (F ⊕ G) f g (inj₂ y) = inj₂ (map G f g y)
map (F ⊗ G) f g (x , y)  = map F f g x , map G f g y
map (F ⊙ G) f g ⟨ x ⟩    = ⟨ map F (map G f g) (map (F ⊙ G) f g) x ⟩
```

To understand the case for composition recall its interpretation: the parameters are G bifunctors, which we handle with map G, and the recursive occurrences are again compositions, so we recursively invoke map (F ⊙ G).

A map over the parameters, pmap, operating on fixed points of bifunctors, can be built from map trivially:

[1] In fact, it is even questionable if polyp encodes true composition, or only a form of functor combination. We will, however, stick to the name "composition" for historical purposes.

```
pmap : {α β : Set} (F : Code) → (α → β) → μ F α → μ F β
pmap F f ⟨ x ⟩ = ⟨ map F f (pmap F f) x ⟩
```

As an example encoding in `polyp` we show the type of non-empty rose trees. We first encode lists in ListC; rose trees are a parameter and a list containing more rose trees (RoseC):

```
ListC : Code
ListC = U ⊕ (P ⊗ I)

RoseC : Code
RoseC = P ⊗ (ListC ⊙ I)
```

The smallest possible rose tree is sRose, containing a single element and an empty list. A larger tree lRose contains a parameter and a list with two small rose trees:

```
sRose : μ RoseC ⊤
sRose = ⟨ tt, ⟨ inj₁ tt ⟩ ⟩

lRose : μ RoseC ⊤
lRose = ⟨ tt, ⟨ inj₂ (sRose, ⟨ inj₂ (sRose, ⟨ inj₁ tt ⟩) ⟩) ⟩ ⟩
```

4.2 Haskell implementation

An earlier encoding of `polyp` directly in Haskell used Template Haskell [Norell and Jansson, 2004]. Nowadays we can encode `polyp` in a way similar to `regular` using type families. We first show the representation types:

```
data U   ρ α = U
data I   ρ α = I {unI :: α}
data P   ρ α = P ρ
data K β ρ α = K β
data (φ :+: ψ) ρ α = L (φ ρ α) | R (ψ ρ α)
data (φ :×: ψ) ρ α = φ ρ α :×: ψ ρ α
data (δ :○: φ) ρ α = Comp (μ δ (φ ρ α))
```

The differences from `regular` are:

- We carry another type parameter around, ρ, which stands for the parameter that `polyp` abstracts over. Types with multiple parameters are supported, but only one parameter is not treated as a constant. So only this parameter ρ can be mapped over, for instance. Recall that `regular` treats all parameters as constants (with the K representation type).
- The parameter ρ is used in the P representation type, and carried around otherwise.

- A new representation type :○: stands for code composition. Its constructor has the same asymmetric treatment of parameters as the Agda model we have shown previously. The fixed-point operator with one parameter in Haskell is:

  ```
  data μ δ ρ = In {out :: δ ρ (μ δ ρ)}
  ```

  Note also that we define both I and μ as records. This is only to simplify later definitions by using the record selector function instead of pattern matching.

4.2.1 Generic functions

As before, generic functions are written by induction over the universe types using a type class. We show a bifunctorial map function as example:

```
class Bimap φ where
  bimap :: (α → β) → (γ → δ) → φ α γ → φ β δ

instance Bimap U where
  bimap _ _ U       = U
instance Bimap P where
  bimap f _ (P x)   = P (f x)
instance Bimap I where
  bimap _ g (I r)   = I (g r)
instance Bimap (K α) where
  bimap _ _ (K x)   = K x
instance (Bimap φ, Bimap ψ) ⇒ Bimap (φ :+: ψ) where
  bimap f g (L x)       = L (bimap f g x)
  bimap f g (R x)       = R (bimap f g x)
instance (Bimap φ, Bimap ψ) ⇒ Bimap (φ :×: ψ) where
  bimap f g (x :×: y)   = bimap f g x :×: bimap f g y
instance (Bimap δ, Bimap φ) ⇒ Bimap (δ :○: φ) where
  bimap f g (Comp x)  = Comp (pmap (bimap f g) x)
```

The case for composition relies on a function to map over the parameters of a fixed point:

```
pmap :: (Bimap δ) ⇒ (α → β) → μ δ α → μ δ β
pmap f = In ∘ bimap f (pmap f) ∘ out
```

When specialised to lists, pmap implements the standard Haskell map. Other generic functions can be defined similarly, either as an algebra (like lengthAlg in Section 3.2.3), or by handling recursion directly in the I instance.

4.2.2 Datatype representation

We again use a type class to convert between datatypes and their generic representation as `polyp` bifunctors:

```
class PolyP (δ :: ⋆ → ⋆) where
  type PF δ :: ⋆ → ⋆ → ⋆
  from      :: δ ρ          → μ (PF δ) ρ
  to        :: μ (PF δ) ρ → δ ρ
```

In the original description the equivalent of this class was called Regular; we rename it to PolyP to avoid confusion with the `regular` library. There are two important differences from Regular:

- The class parameter δ has kind $\star \to \star$. So we represent only the container types, such as lists. Types of kind $\star$ cannot be directly represented in this class; in Section 11.2 we show how to support both kinds of datatype with a single set of representation types.
- To be more faithful to the original description, we opt for a deep encoding, unlike the shallow encoding of `regular`. So the function from, for instance, takes a user datatype to a fixed point of a `polyp` bifunctor.

This deep encoding also requires the conversion functions to be directly recursive. As an example, we show how to encode the standard list type:

```
instance PolyP [] where
  type PF [] = U :+: (P :×: I)
  from []       = In (L U)
  from (h : t)  = In (R (P h :×: I (from t)))
  to (In (L U))              = []
  to (In (R (P h :×: I t)))  = h : to t
```

To see an example with composition, consider the type of rose trees:

```
data Rose ρ = Rose ρ [Rose ρ]
```

Its representation in `polyp` follows:

```
instance PolyP Rose where
  type PF Rose = P :×: (PF [] :○: I)
  from (Rose x l) = In (P x :×: Comp (pmap (I ∘ from) (from l)))
  to (In (P x :×: Comp l)) = Rose x (to (pmap (to ∘ unI) l))
```

Note that we use PF [] (and not []) in the pattern functor for Rose. Consequently we need to use pmap to map over the lists of rose trees, also relying on the fact that [] is an instance of PolyP.

The `polyp` pmap function on lists corresponds to the standard Haskell map function. We can write a more user-friendly interface to pmap to take care of the necessary conversions:

```
gfmap :: (PolyP δ, Bimap (PF δ)) ⇒ (α → β) → δ α → δ β
gfmap f = to ∘ pmap f ∘ from
```

We call this function gfmap as it is the generic variant of the fmap function from the standard Functor class. As an example, gfmap (+1) [1, 2, 3] evaluates to [2, 3, 4].

We have seen that `polyp` generalises `regular` to add support for one datatype parameter. In the next chapter we continue to increase the flexibility of generic programming approaches, this time by considering how to support families of mutually recursive datatypes.

CHAPTER 5

The `multirec` library

Like `polyp`, the `multirec` library [Rodriguez Yakushev et al., 2009] is also a generalisation of `regular`, allowing for multiple recursive positions instead of only one. This means that families of mutually recursive datatypes can be encoded in `multirec`. While both `regular` and `polyp` deal with mutual recursion by encoding datatypes (other than that being defined) as constants, in `multirec` the family is defined as a single group, allowing for catamorphisms over families, for instance.

5.1 Agda model

In `multirec`, types are represented as higher-order (or indexed) functors:

```
Indexed : Set → Set₁
Indexed α = α → Set
```

Codes themselves are parametrised over an index Set, that is used in the I case. Furthermore, we have a new code ! for tagging a code with a particular index:

```
data Code (I : Set) : Set where
  U     : Code I
  I     : I                → Code I
  !     : I                → Code I
  _⊕_ : (F G : Code I) → Code I
  _⊗_ : (F G : Code I) → Code I
```

Since `multirec` does not support parameters, there is no code for composition.

The interpretation is parametrised by a function r that maps indices (recursive positions) to Sets, and a specific index i that defines which particular position we are interested in; since a code can define several types, ⟦_⟧ is a function from the index of a particular type to its interpretation. For an occurrence of an index I we retrieve the associated set using the function r. Tagging constrains the interpretation to a particular index j, so its interpretation is index equality:

```
⟦_⟧ : {I : Set} → Code I → Indexed I → Indexed I
⟦ U     ⟧ r i = ⊤
⟦ I j   ⟧ r i = r j
⟦ ! j   ⟧ r i = i ≡ j
⟦ F ⊕ G ⟧ r i = ⟦ F ⟧ r i ⊎ ⟦ G ⟧ r i
⟦ F ⊗ G ⟧ r i = ⟦ F ⟧ r i × ⟦ G ⟧ r i
```

Mapping is entirely similar to the `regular` map, only that the function being mapped is now an index-preserving map. An index-preserving map is a transformation between indexed functors that does not change their index:

```
_⇉_ : {I : Set} → Indexed I → Indexed I → Set
r ⇉ s = ∀ i → r i → s i
```

The map function is then an index-preserving map over the interpretation of a code:

```
map : {I : Set} {r s : Indexed I} (F : Code I) → r ⇉ s → ⟦ F ⟧ r ⇉ ⟦ F ⟧ s
map U       f i tt       = tt
map (I j)   f i x        = f j x
map (! j)   f i x        = x
map (F ⊕ G) f i (inj₁ x) = inj₁ (map F f i x)
map (F ⊕ G) f i (inj₂ y) = inj₂ (map G f i y)
map (F ⊗ G) f i (x , y)  = map F f i x , map G f i y
```

We also need an indexed variant of the fixed-point operator:

```
data μ {I : Set} (F : Code I) (i : I) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) i → μ F i
```

To show an example involving mutually recursive types we encode a zig-zag sequence of even length. Consider first the family we wish to encode, inside a **mutual** block as the datatypes are mutually recursive:

```
mutual
  data Zig : Set where
    zig : Zag → Zig
    end : Zig
  data Zag : Set where
    zag : Zig → Zag
```

We can encode this family in `multirec` as follows:

```
ZigC : Code (⊤ ⊎ ⊤)
ZigC = I (inj₂ tt) ⊕ U

ZagC : Code (⊤ ⊎ ⊤)
ZagC = I (inj₁ tt)

ZigZagC : Code (⊤ ⊎ ⊤)
ZigZagC = (! (inj₁ tt) ⊗ ZigC) ⊕ (! (inj₂ tt) ⊗ ZagC)

zigZagEnd : μ ZigZagC (inj₁ tt)
zigZagEnd = ⟨ inj₁ (refl , inj₁ ⟨ inj₂ (refl , ⟨ inj₁ (refl , inj₂ tt) ⟩) ⟩) ⟩
```

zigZagEnd encodes the value zig (zag end), as its name suggests. Note how we define the code for each type in the family separately (ZigC and ZagC), and then a code ZigZagC for the family, encoding a tagged choice between the two types. As a consequence, proofs of index equality (witnessed by refl) are present throughout the encoded values.

5.2 Haskell implementation

The encoding of `multirec` in Haskell makes use of more advanced type features than are necessary for `regular`. Again we have a number of representation datatypes, now parametrised over an indexed recursion point argument ρ and an index ι:

```
data U       (ρ :: ⋆ → ⋆) ι = U
data I κ     (ρ :: ⋆ → ⋆) ι = I {unI :: ρ κ}
data K α     (ρ :: ⋆ → ⋆) ι = K {unK :: α}
data (ϕ :+: ψ) (ρ :: ⋆ → ⋆) ι = L (ϕ ρ ι) | R (ψ ρ ι)
data (ϕ :×: ψ) (ρ :: ⋆ → ⋆) ι = ϕ ρ ι :×: ψ ρ ι
```

This is a standard way of encoding mutual recursion as higher-order fixed points. The ρ argument is used as a selector of which type to recurse into; in the I case for recursive occurrences, it is used with a specific index κ.

The ι argument has a different purpose. Since we are encoding multiple datatypes in a single datatype, we use ι to focus on one particular type of the family. For focusing we use the $\triangleright$ representation type, analogous to the ! code of the Agda model:

```
data ϕ ▷ ι :: (⋆ → ⋆) → ⋆ → ⋆ where
  Tag :: ϕ ρ ι → (ϕ ▷ ι) ρ ι
```

We write $\triangleright$ as a generalised algebraic datatype (GADT) [Schrijvers et al., 2009] to introduce a type-level equality constraint (equivalent to the _≡_ Agda type), signalled by the appearance of ι twice in the result type of Tag. This means that Tag can only be built (or type checked) when the requested index is the same as that given as argument to $\triangleright$. The example datatype encoding in Section 5.2.2 makes the use of tagging more clear.

5.2.1 Generic functions

As usual, generic functions are defined by giving instances of a type class for each representation type. We show the higher-order mapping function for `multirec`:

```
class HFunctor ξ ϕ where
  hmap :: (forall ι. ξ ι → ρ ι → σ ι) → ξ ι → ϕ ρ ι → ϕ σ ι
```

Its type in Haskell is similar to the type in Agda, transforming the recursive points of a representation type ϕ from ρ to σ while keeping the index ι unchanged. Additionally, we take a function to apply in the recursive occurrences, similarly to `regular`, as `multirec` implements a shallow view on data. Finally, the argument of type $\xi\ \iota$, which also occurs in the function for the recursive case, stands for the family index. The instance of HFunctor for I illustrates its use:

```
class El ξ ι where
  proof :: ξ ι

instance El ξ ι ⇒ HFunctor ξ (I ι) where
  hmap f _ (I x) = I (f proof x)
```

For a recursive occurrence under I we simply call the function we carry around. This function takes as first argument an index for which type in the family it is called on. To simplify the construction of such indices we also define class El; an instance of El $\xi\ \iota$ means that ι is an index of the family ξ, evidenced by the element proof of type $\xi\ \iota$. In Section 5.2.2 we show how to instantiate El for an example family, to further help clarify its use.

The remaining cases are simple; we recurse through sums, products, and tags, and return in units and constants:

```
instance HFunctor ξ U where
  hmap _ _ U       = U
instance HFunctor ξ (K α) where
  hmap _ _ (K x)   = K x
instance (HFunctor ξ ϕ, HFunctor ξ ψ) ⇒ HFunctor ξ (ϕ :+: ψ) where
  hmap f  p (L  x) = L  (hmap f p x)
  hmap f  p (R x)  = R  (hmap f p x)
instance (HFunctor ξ ϕ, HFunctor ξ ψ) ⇒ HFunctor ξ (ϕ :×: ψ) where
  hmap f  p (x :×: y) = hmap f p x :×: hmap f p y
instance HFunctor ξ ϕ ⇒ HFunctor ξ (ϕ :>: ι) where
  hmap f  p (Tag x) = Tag (hmap f p x)
```

We can use hmap to define a higher-order catamorphism function, but we show first how to represent datatypes.

5.2.2 Datatype representation

Families of datatypes are represented by instances of the El class shown before, and also the Fam class:

```
newtype I0 α = I0 α

class Fam ξ where
  type PF (ξ :: ⋆ → ⋆) :: (⋆ → ⋆) → ⋆ → ⋆
  from :: ξ ι → ι       → PF ξ I0 ι
  to   :: ξ ι → PF ξ I0 ι → ι
```

In Fam we have the usual from and to conversion functions, and the pattern functor representation as a type family. The pattern functor takes the family ξ as an argument, and is indexed over the recursive point argument (which we called ρ when defining the representation types) and a particular index ι. The conversion functions instantiate ρ to I_0, a type-level identity.

We revisit our zig-zag mutually recursive datatypes in Haskell to show an example instantiation:

```
data Zig = Zig Zag | End
data Zag = Zag Zig
```

The first step to encode these in `multirec` is to define a datatype to represent the family:

```
data ZigZag ι where
  ZigZagZig :: ZigZag Zig
  ZigZagZag :: ZigZag Zag
```

ZigZag now stands for the family type, with $\text{ZigZag}_{\text{Zig}}$ and $\text{ZigZag}_{\text{Zag}}$ being respectively the evidence for the presence of Zig and Zag in this family. This is encoded in the instance of El:

```
instance El ZigZag Zig where
  proof = ZigZagZig
instance El ZigZag Zag where
  proof = ZigZagZag
```

We are left with the instance of Fam:

```
instance Fam ZigZag where
  type PF ZigZag = ((I Zag :+: U) :>: Zig)
                 :+: (I Zig       :>: Zag)
  from ZigZagZig (Zig zag) = L (Tag (L (I (I0 zag))))
```

```
from ZigZag_Zig  End          =  L (Tag (R U))
from ZigZag_Zag  (Zag zig)    =  R (Tag (I (I_0 zig)))
to ZigZag_Zig  (L (Tag (L (I (I_0 zag)))))  =  Zig zag
to ZigZag_Zig  (L (Tag (R U)))              =  End
to ZigZag_Zag  (R (Tag (I (I_0 zig))))      =  Zag zig
```

Since from and to operate on multiple datatypes we use their first argument, the family proof, to set which type we are converting. Pattern matching on the proof is also what allows us to produce the Tag constructor, as its type requires that the value being constructed matches the index declared in the tag representation type $\gg$.

Using the Fam class we can define a higher-order catamorphism function that takes a user datatype, converts it to the generic representation, and applies an algebra recursively using hmap:

```
type Algebra ξ ρ  =  forall ι. ξ ι → (PF ξ) ρ ι → ρ ι
cata :: (Fam ξ, HFunctor ξ (PF ξ))  ⇒  Algebra ξ ρ → ξ ι → ι → ρ ι
cata f p  =  f p ∘ hmap (λ p (I_0 x) → cata f p x) p ∘ from p
```

Note that this catamorphism does not have to return a single type for a family. Its return type is $\rho\ \iota$, implying that different types can be returned for different family indices. A generic length function for a family has a single return type Int, but an identity catamorphism does not, for instance. More realistic examples include the evaluation of an abstract syntax tree, which is shown as an example in the source distribution of `multirec`.[1]

We have seen that adding support for families of datatypes requires lifting the indices from Sets to Set-indexed functions. The reader might wonder if something similar can be done to lift `polyp`'s restriction to a single parameter. That has been investigated before [Hesselink, 2009], with mixed results: while it is possible, the resulting Haskell library is hard to use, and there is no support for family composition. In the next chapter we look at a different approach that does allow code composition.

[1] `http://hackage.haskell.org/package/multirec`

CHAPTER 6

Indexed functors

This chapter presents `indexed`, a powerful approach to generic programming in Agda using indexed functors. Unlike the previous approaches we have seen, its universe incorporates fixed points, and also supports composition, indexing, and isomorphisms.

We study this approch in more detail since it generalises all of the previous approaches. We provide proofs for the functor laws (Section 6.2), define a decidable equality function (Section 6.3), and show a zipper for indexed functors (Section 6.4). We have not shown these for the previous libraries, but we show in Chapter 8 how the previous libraries can use the facilities provided by `indexed`.

6.1 A universe for indexed functors

The `indexed` approach is centered on the notion of an indexed functor. A normal functor is a type constructor, i.e. of Agda type Set → Set. An indexed functor allows an indexed set both as input and as output, where an indexed set is a function mapping a concrete "index" type to Set. Similarly to `multirec`, families of mutually recursive datatypes can be expressed as a single indexed functor, by choosing appropriate indices. Parametrised datatypes are represented using indices as well; in this way recursion and parametrisation are treated uniformly, allowing for flexible composition. Similarly to the universes of Altenkirch et al. [2007], `indexed` includes the fixed-point operator within the universe, which simplifies code reuse. To allow for easy encapsulation of user-defined datatypes, datatype isomorphisms are part of the universe itself.

The result is a very general universe that can encode many more datatypes than the previous approaches we have seen. Unlike Chapman et al. [2010], however, `indexed` is

not a minimal, self-encoding universe, but instead a representation that maps naturally to the usual way of defining a datatype.

Unlike the libraries we have seen so far, the `indexed` approach was originally described in Agda [Löh and Magalhães, 2011]. An added advantage of working in a dependently typed setting is that properties of generic functions are just generic functions themselves, and can be proven within the same framework. We show how indexed functors adhere to the functor laws. As examples of how to work with the universe, we derive the catamorphism and other recursion schemes. Along with all the functionality that can be defined using the recursive morphisms, we show that our approach also allows defining functions by direct induction on the codes, and show decidable equality as an example. Finally, we present the basic ingredients for navigation of datatypes described as indexed functors using a zipper [Huet, 1997], relying on an underlying derivative [McBride, 2001] for the universe.

6.1.1 Basic codes

The codes of the universe describe indexed functors, and are therefore parametrised over two sets. Intuitively, these can be thought of as the indices for inputs and outputs of the type described by the code, so we generally call the first argument I and the second O. A code I ▶ O describes a type that is parametrised over inputs from I and that is itself indexed over O. In many standard examples, I and O will be instantiated to finite types. We get a classic functor of type Set → Set back by instantiating both I and O with the one-element type ⊤. In case we want a code to represent a family of n datatypes, we can instantiate O to the type Fin n that has n elements.

Since the `indexed` universe is large, we present it in an incremental fashion, starting with the base cases:

```
data _▶_ (I : Set) (O : Set) : Set₁ where
  Z    : I ▶ O
  U    : I ▶ O
```

The codes are a relation between two sets, so we use an infix operator for the universe type, _▶_, taking the input set on the left and the output set on the right. The constructor Z is used for empty types, and U for unit types. Both base codes are polymorphic on the input and output indices. We do not use a code K for inclusion of arbitrary constant types, as these prove troublesome when defining generic functions. Instead we use isomorphisms, as shown later in this chapter.

Disjoint sum, product and composition are used for combining codes:

```
_⊕_ :                  I ▶ O → I ▶ O → I ▶ O
_⊗_ :                  I ▶ O → I ▶ O → I ▶ O
_◎_ : {M : Set} → M ▶ O → I ▶ M → I ▶ O
```

Sum (_⊕_) and product (_⊗_) combine two codes with input I and output O to produce a code I ▶ O. For a composition F ◎ G we require the codes to combine to be compatible:

the output of code G needs to be the same as the input of the code F (namely M). We connect the output of G to the input of F, and produce a code with the input of G and the output of F.

Together with the universe, we define an interpretation function that establishes the relation between codes and Agda types. A code I ▶ O is interpreted as a function between indexed functors I ▷ O:

```
Indexed : Set → Set₁
Indexed I = I → Set

_▷_ : Set → Set → Set₁
I ▷ O = Indexed I → Indexed O
```

When defining the actual interpretation function ⟦_⟧, we thus have a parameter r : I → Set describing all the input indices and a parameter o : O selecting a particular output index available:

```
⟦_⟧ : {I O : Set} → I ▶ O → I ▷ O
⟦ Z     ⟧ r o = ⊥
⟦ U     ⟧ r o = ⊤
⟦ F ⊕ G ⟧ r o = ⟦ F ⟧ r o ⊎ ⟦ G ⟧ r o
⟦ F ⊗ G ⟧ r o = ⟦ F ⟧ r o × ⟦ G ⟧ r o
⟦ F ⊙ G ⟧ r o = (⟦ F ⟧ ∘ ⟦ G ⟧) r o
```

We interpret Z as the empty type ⊥ (with no constructors), and U as the singleton type ⊤. Each of the three constructors for sum, product, and composition is interpreted as the corresponding concept on Agda types.

With this part of the universe in place we can already encode some simple, non-recursive datatypes. Consider the type of Boolean values Bool with constructors true and false:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

It is a single, non-indexed datatype, which takes no parameters and is not recursive. We can encode it as an indexed functor using a type with zero inhabitants for I and a type with one inhabitant for O:

```
'BoolC' : ⊥ ▶ ⊤
'BoolC' = U ⊕ U
```

To convert between the type of Booleans and its representation we use two functions:

```
fromBool : Bool → ⟦ 'BoolC' ⟧ (λ ()) tt
fromBool true  = inj₁ tt
fromBool false = inj₂ tt
```

```
toBool : ⟦ 'BoolC' ⟧ (λ ()) tt → Bool
toBool (inj₁ tt) = true
toBool (inj₂ tt) = false
```

We instantiate the input indexed set to (λ ()), the absurd match. Recall that its type is Indexed ⊥, or ⊥ → Set. A function that takes an uninhabited type as an argument will never be called, so to define it we simply have to match on the empty type with the absurd pattern (). The output index is set to tt, as this family only has a single output index.

6.1.2 Isomorphisms

Being in a dependently-typed language, we can also provide the proof that fromBool and toBool indeed form an isomorphism:

```
isoBool₁ : ∀ b → toBool (fromBool b) ≡ b
isoBool₁ true  = refl
isoBool₁ false = refl
isoBool₂ : ∀ b → fromBool (toBool b) ≡ b
isoBool₂ (inj₁ tt) = refl
isoBool₂ (inj₂ tt) = refl
```

For convenience, we wrap all the above in a single record, which we introduced in Section 2.2.3 but repeat here for convenience:

```
infix 3 _≃_
record _≃_ (A B : Set) : Set where
  field
    from : A → B
    to   : B → A
    iso₁ : ∀ x → to (from x) ≡ x
    iso₂ : ∀ x → from (to x) ≡ x
```

The particular instantiation for Bool simply uses the functions we have defined before:

```
isoBool : (r : Indexed ⊥) (o : ⊤) → Bool ≃ ⟦ 'BoolC' ⟧ r o
isoBool r o = record { from = fromBool
                     ; to   = toBool
                     ; iso₁ = isoBool₁
                     ; iso₂ = isoBool₂ }
```

Isomorphisms are useful when we want to view a type as a different, but equivalent type. In the case for Bool and ⟦ 'BoolC' ⟧, for instance, the former is more convenient to use, but the latter can be used generically. If we add isomorphisms to the universe, generic functions can automatically convert between a user type and its representation:

```
Iso : (C : I ▶ O) → (D : I ▷ O) →
      ((r : Indexed I) → (o : O) → D r o ≃ ⟦ C ⟧ r o) → I ▶ O
```

The code Iso captures an isomorphism between the interpretation of a code C and an indexed functor D. This indexed functor D is used in the interpretation of the code Iso:

```
⟦ Iso C D e ⟧ r o = D r o
```

Having Iso has two advantages. First, it enables having actual user-defined Agda datatypes in the image of the interpretation function. For example, we can give a code for the Agda datatype Bool now:

```
'Bool' : ⊥ ▶ ⊤
'Bool' = Iso 'BoolC' (λ _ _ → Bool) isoBool
```

Secondly, Iso makes it possible to reuse previously defined types inside more complex definitions, while preserving the original code for further generic operations—in Section 6.1.7 we show an example of this technique by reusing the code for natural numbers inside the definition of a simple language.

Note that the Iso code is not specific to `indexed` only; the previous Agda models could have a code for isomorphisms as well. But since this code does not add expressiveness to the universe, and the models are a concise description of each library, we have not added isomorphisms to them.

6.1.3 Adding fixed points

To encode recursive datatypes we need some form of fixed-point operator. In the Agda models we have seen so far this operator is a separate datatype, outside the universe. Indexed functors, however, have the nice property of being closed under fixed points: if we have an indexed functor where the recursive calls come from the same index set as the output, i.e. a functor O ▷ O, then its fixed point will be of type ⊥ ▷ O. If we consider a functor with parameters of type I ⊎ O ▷ O, with input indices being either parameters from I, or recursive calls from O, we can obtain a fixed point of type I ▷ O.

Since the fixed point of an indexed functor is itself an indexed functor, it is convenient to just add fixed points as another constructor to the type of codes:

```
Fix : (I ⊎ O) ▶ O → I ▶ O
```

As indicated before, the code Fix transforms a code with I ⊎ O input indices and O output indices into a code of type I ▶ O. Naturally, we still need a way to actually access inputs. For this we use the I code, similarly to the previous approaches:

```
I  : I          → I ▶ O
```

Using I we can select a particular input index (which, in the light of Fix, might be either a parameter or a recursive call).

We use a datatype μ with a single constructor $\langle\_\rangle$ to generate the interpretation of Fix F from that of F. We know that F : (I ⊎ O) ▶ O, so the first argument to ⟦ F ⟧ needs to discriminate between parameters and recursive occurrences. We use r | μ F r for this purpose, i.e. for parameters we use r : I → Set, whereas recursive occurrences are interpreted with μ F r : O → Set:

```
_|_ : {I J : Set} → Indexed I → Indexed J → Indexed (I ⊎ J)
(r | s) (inj₁ i) = r i
(r | s) (inj₂ j) = s j

mutual
  data μ {I O : Set} (F : (I ⊎ O) ▶ O) (r : Indexed I) (o : O) : Set where
    ⟨_⟩ : ⟦ F ⟧ (r | μ F r) o → μ F r o
...
⟦ Fix F ⟧ r o = μ F r o
⟦ I i ⟧   r o = r i
```

The interpretation of I uniformly invokes r for every input index i.

As an example of a datatype with parameters and recursion, we show how to encode parametric lists (introduced in Section 2.1). We start by encoding the base functor of lists:

```
'ListC' : (⊤ ⊎ ⊤) ▶ ⊤
'ListC' = U ⊕ (I (inj₁ tt) ⊗ I (inj₂ tt))
```

The arguments of _▶_ reflect that we are defining one type (output index ⊤) with two inputs (input index ⊤ ⊎ ⊤), where one represents the parameter and the other represents the recursive call.

We use a product to encode the arguments to the _::_ constructor. The first argument is an occurrence of the first (and only) parameter, and the second is a recursive occurrence of the first (and only) type being defined (namely list).

Using Fix, we can now close the recursive gap in the representation of lists:

```
'ListF' : ⊤ ▶ ⊤
'ListF' = Fix 'ListC'
```

We can confirm that our representation is isomorphic to the original type by providing conversion functions:

```
fromList : ∀ {r o} → [ r o ] → ⟦ 'ListF' ⟧ r o
fromList []       = ⟨ inj₁ tt ⟩
fromList (x :: xs) = ⟨ inj₂ (x , fromList xs) ⟩

toList : ∀ {r o} → ⟦ 'ListF' ⟧ r o → [ r o ]
toList ⟨ inj₁ tt ⟩       = []
toList ⟨ inj₂ (x , xs) ⟩ = x :: toList xs
```

As before, we can show that the conversion functions really form an isomorphism. The isomorphism is entirely trivial, but the implementation is a bit cumbersome because we have to explicitly pass some implicit arguments in order to satisfy Agda's typechecker:

```
isoList₁ : ∀ {r o} → (l : [ r o ]) → toList {r} (fromList l) ≡ l
isoList₁      []       = refl
isoList₁ {r} (h :: t)  = cong (λ x → h :: x) (isoList₁ {r} t)

isoList₂ : ∀ {r o} → (l : ⟦ 'ListF' ⟧ r o) → fromList (toList l) ≡ l
isoList₂ ⟨ inj₁ tt ⟩      = refl
isoList₂ ⟨ inj₂ (h , t) ⟩ = cong (λ x → ⟨ inj₂ (h , x) ⟩) (isoList₂ t)

isoList : ∀ {r o} → [ r o ] ≃ ⟦ 'ListF' ⟧ r o
isoList {r} = record {from = fromList
                      ; to   = toList
                      ; iso₁ = isoList₁ {r}
                      ; iso₂ = isoList₂}
```

For conciseness we will refrain from showing further proofs of isomorphisms.

We now define a code for lists including the isomorphism between $[\_]$ and μ 'ListC':

```
'List' : ⊤ ▶ ⊤
'List' = Iso (Fix 'ListC') (λ f t → [ f t ]) (λ r o → isoList)
```

This code makes it possible for us to use actual Agda lists in generic operations, and explicit applications of the conversion functions toList and fromList are no longer necessary.

Having Fix in the universe (as opposed to using it externally) has the advantage that codes become more reusable. For example, we can reuse the code for lists we have just defined when defining a code for rose trees (Section 6.1.6). This is an instance of the common situation where a fixed point is used as the first argument of a composition. Therefore `indexed` allows encoding datatypes that involve several applications of Fix.

6.1.4 Mapping indexed functors

To show that `indexed` codes are interpreted as indexed functors, we define a map operation on codes.

Since we are working with indexed sets rather than sets, we have to look at arrows between indexed sets, which are index-preserving mappings:

```
_⇉_ : {I : Set} → Indexed I → Indexed I → Set
r ⇉ s = ∀ i → r i → s i
```

As in `multirec`, map lifts such an index-preserving function r ⇉ s between two indexed sets r and s to an indexed-preserving function ⟦ C ⟧ r ⇉ ⟦ C ⟧ s on the interpretation of a code C:

```
map : {I O : Set} {r s : Indexed I} →
      (C : I ▶ O) → r ⇉ s → ⟦ C ⟧ r ⇉ ⟦ C ⟧ s
```

Note that if we choose $I = O = \top$, the indexed sets become isomorphic to sets, and the index-preserving functions become isomorphic to functions between two sets, i.e. we specialise to the well-known Haskell-like setting of functors of type Set → Set.

Let us look at the implementation of map:

```
map Z     f o ()
map U     f o tt  = tt
map (I i) f o x   = f i x
```

If we expand the type of map we see that it takes four explicit arguments. The first two are the code and the index-preserving function f. The type of the function returned by map can be expanded to ∀ o → ⟦ C ⟧ r o → ⟦ C ⟧ s o, explaining the remaining two arguments: an arbitrary output index o, and an element of the interpreted code.

The interpretation of code Z has no inhabitants, so we do not have to give an implementation. For U, we receive a value of type ⊤, which must be tt, and in particular does not contain any elements, so we return tt unchanged. On I we get an element x corresponding to input index i, which we supply to the function f at index i.

For sums and products we keep the structure, pushing the map inside. Composition is handled with a nested map, and for fixed points we adapt the function argument to take into account the two different types of indices; using an auxiliary _||_ operator to merge index-preserving mappings, left-tagged indices (parameters) are mapped with f, whereas right-tagged indices (recursive occurrences) are mapped recursively:

```
map (F ⊕ G) f o (inj₁ x)  = inj₁ (map F  f o x)
map (F ⊕ G) f o (inj₂ x)  = inj₂ (map G f o x)
map (F ⊗ G) f o (x , y)   = map F f o x , map G f o y
map (F ◎ G) f o x         = map F (map G f) o x
map (Fix F)   f o ⟨ x ⟩   = ⟨ map F (f || map (Fix F) f) o x ⟩
```

The operator for merging index-preserving mappings, used in the fixed point case, follows:

```
_||_ : {I J : Set} {r u : Indexed I} {s v : Indexed J} →
       r ⇉ u → s ⇉ v → (r | s) ⇉ (u | v)
(f || g) (inj₁ x)  =  f x
(f || g) (inj₂ x)  =  g x
```

We are left with a case for isomorphisms to complete our definition of map, which we define with judicious use of the conversion functions:

```
map {r = r} {s = s} (Iso C D e) f o x with (e r o , e s o)
... | ep₁ , ep₂  =  _≃_.to ep₂ (map C f o (_≃_.from ep₁ x))
```

We use Agda's **with** construct to pattern-match on the embedding-projection pairs, from which we extract the necessary conversion functions.

As an example, let us look at the instance of map on lists. We obtain it by specialising the types:

```
mapList : {A B : Set} → (A → B) → [ A ] → [ B ]
mapList f = map ‘List’ (const f) tt
```

We are in the situation described before, where both index sets are instantiated to ⊤. Note that we do not need to apply any conversion functions, since ‘List’ contains the isomorphism between lists and their representation. We discuss how to prove the functor laws for map in Section 6.2.

For now, we can confirm that mapList works as expected on a simple example:

```
mapListExample : mapList su (1 :: 2 :: []) ≡ (2 :: 3 :: [])
mapListExample = refl
```

Since this code typechecks, we know that mapList su (1 :: 2 :: []) indeed evaluates to 2 :: 3 :: [].

6.1.5 Recursion schemes

Equipped with a mapping function for indexed functors, we can define basic recursive morphisms in a conventional fashion:

```
id⇉ : {I : Set} {r : Indexed I} → r ⇉ r
id⇉ i = id
cata : {I O : Set} {r : Indexed I} {s : Indexed O} →
       (C : (I ⊎ O) ▶ O) → ⟦ C ⟧ (r | s) ⇉ s → ⟦ Fix C ⟧ r ⇉ s
cata C φ i ⟨ x ⟩ = φ i (map C (id⇉ || cata C φ) i x)
```

The catamorphism on indexed functors is not much different from the standard functorial catamorphism. The important difference is the handling of left and right indices: since we wish to traverse over the structure only, we apply the identity on indexed sets $\mathsf{id}_{⇉}$ to parameters, and recursively apply cata to right-tagged indices.

As an example, let us consider lists again and see how the foldr function can be expressed in terms of the generic catamorphism:

```
_▽_ : {A B C : Set} → (A → C) → (B → C) → (A ⊎ B) → C
(r ▽ s) (inj₁ i) = r i
(r ▽ s) (inj₂ j) = s j
foldr : {A B : Set} → (A → B → B) → B → [ A ] → B
foldr {A} c n l = cata {r = const A} ‘ListC’ φ tt (fromList l)
                  where φ = const (const n ▽ uncurry c)
```

The function foldr invokes cata. The parameters are instantiated with const A, i.e. there is a single parameter and it is A, and the recursive slots are instantiated with const B, i.e. there is a single recursive position and it will be transformed into a B. We invoke the catamorphism on 'ListF', which means we have to manually apply the conversion fromList on the final argument to get from the user-defined list type to the isomorphic structural representation. Ultimately we are interested in the single output index tt that lists provide.

This leaves the algebra φ. The generic catamorphism expects an argument of type $[\![$'ListF' $]\!]$ (r | s) $\rightrightarrows$ s, which in this context reduces to (i : $\top$) $\to$ $\top \uplus$ (A $\times$ B) $\to$ B. On the other hand, foldr takes the nil- and cons- components separately, which we can join together with _$\triangledown$_ to obtain something of type $\top \uplus$ (A $\times$ B) $\to$ B. We use const to ignore the trivial index.

We can now define the length of a list using foldr and check that it works as expected:

```
length : {A : Set} → [ A ] → ℕ
length = foldr (const su) ze

lengthExample : length (1 :: 0 :: []) ≡ 2
lengthExample = refl
```

Many other recursive patterns can be defined similarly. We show the definitions of anamorphism and hylomorphism:

```
ana : {I O : Set} {r : Indexed I} {s : Indexed O} →
      (C : (I ⊎ O) ▶ O) → s ⇉ ⟦ C ⟧ (r | s) → s ⇉ ⟦ Fix C ⟧ r
ana C ψ i x = ⟨ map C (id⇉ || ana C ψ) i (ψ i x) ⟩

hylo : {I O : Set} {r : Indexed I} {s t : Indexed O} →
       (C : (I ⊎ O) ▶ O) → ⟦ C ⟧ (r | t) ⇉ t → s ⇉ ⟦ C ⟧ (r | s) → s ⇉ t
hylo C φ ψ i x = φ i (map C (id⇉ || hylo C φ ψ) i (ψ i x))
```

6.1.6 Using composition

To show how to use composition we encode the type of rose trees:

```
data Rose (A : Set) : Set where
  fork : A → [ Rose A ] → Rose A
```

The second argument to fork, of type [Rose A], is encoded using composition:

```
'RoseC' : (⊤ ⊎ ⊤) ▶ ⊤
'RoseC' = I (inj₁ tt) ⊗ ('ListF' ⊙ I (inj₂ tt))

'RoseF' : ⊤ ▶ ⊤
'RoseF' = Fix 'RoseC'
```

Note that the first argument of composition here is 'ListF', not 'ListC'. We thus really make use of the fact that fixed points are part of the universe and can appear anywhere within a code. We also show that codes can be reused in the definitions of other codes.

The conversion functions for rose trees follow:

```
fromRose : {r : Indexed ⊤} {o : ⊤} → Rose (r o) → ⟦ 'RoseF' ⟧ r o
fromRose (fork x xs) = ⟨ x , map 'ListF' (λ i → fromRose) tt (fromList xs) ⟩

toRose : {r : Indexed ⊤} {o : ⊤} → ⟦ 'RoseF' ⟧ r o → Rose (r o)
toRose ⟨ x , xs ⟩ = fork x (toList (map 'ListF' (λ i → toRose) tt xs))
```

The use of composition in the code implies the use of map in the conversion functions, since we have to map the conversion over the elements of the list. This also means that to provide the isomorphism proofs for Rose we need to have proofs for the behavior of map. We describe these in detail in Section 6.2.

6.1.7 Parametrised families of datatypes

As an example of the full power of abstraction of indexed functors we show the representation of a family of mutually recursive parametrised datatypes. Our family represents the Abstract Syntax Tree (AST) of a simple language:

```
mutual
  data Expr (A : Set) : Set where
    econst : ℕ                → Expr A
    add    : Expr A → Expr A  → Expr A
    evar   : A                → Expr A
    elet   : Decl A → Expr A  → Expr A
  data Decl (A : Set) : Set where
    assign : A          → Expr A → Decl A
    seq    : Decl A → Decl A     → Decl A
```

In our AST, an expression can be either a natural number constant, the addition of two expressions, a variable, or a let declaration. Declarations are either an assignment of an expression to a variable, or a pair of declarations.

We can easily encode each of the datatypes as indexed functors. We start by defining a type synonym for the output indices, for convenience:

```
AST : Set
AST = ⊤ ⊎ ⊤

expr : AST
expr = inj₁ tt

decl : AST
decl = inj₂ tt
```

Since we are defining two datatypes, we use a type with two inhabitants, namely ⊤ ⊎ ⊤. Note that any other two-element type such as Bool or Fin 2 would also do. We

define expr and decl as shorthands for each of the indices. We can now encode each of the types:

```
‘ExprC’ : (⊤ ⊎ AST) ▶ AST
‘ExprC’ = ?0
        ⊕ I (inj2 expr) ⊗ I (inj2 expr)
        ⊕ I (inj1 tt)
        ⊕ I (inj2 decl) ⊗ I (inj2 expr)
‘DeclC’ : (⊤ ⊎ AST) ▶ AST
‘DeclC’ = I (inj1 tt)     ⊗ I (inj2 expr)
        ⊕ I (inj2 decl) ⊗ I (inj2 decl)
```

Our codes have type (⊤ ⊎ AST) ▶ AST, since we have one parameter (the type of the variables) and two datatypes in the family. For expressions we want to reuse a previously defined ‘ℕ’ code for ℕ (which we do not show). However, ‘ℕ’ has type ⊥ ▶ ⊤, which is not compatible with the current code, so we cannot simply enter ‘ℕ’ in the $?_0$ hole.

We need some form of re-indexing operation to plug indexed functors within each other. Therefore we add the following operation to our universe:

```
_↗_↘_ : {I' O' : Set} → I' ▶ O' → (I' → I) → (O → O') → I ▶ O
```

Now we can fill the hole $?_0$ with the expression ‘ℕ’ ↗ (λ ()) ↘ const tt. The interpretation of this new code is relatively simple. For the input, we compose the re-indexing function with r, and for the output we apply the function to the output index:

```
⟦ F ↗ f ↘ g ⟧ r o = ⟦ F ⟧ (r ∘ f) (g o)
```

Mapping over a re-indexed code is also straightforward:

```
map (F ↗ g ↘ h) f o x = map F (f ∘ g) (h o) x
```

Finally, we can join the two codes for expressions and declarations into a single code for the whole family. For this we need an additional code to specify that we are defining one particular output index, similarly to `multirec`:

```
! : O → I ▶ O
```

The code ! is parametrised by a particular output index. Its interpretation introduces the constraint that the argument index should be the same as the output index we select when interpreting:

```
⟦ ! o' ⟧ r o = o ≡ o'
```

Its usefulness becomes clear when combining the codes for the AST family:

```
‘ASTC’ : (⊤ ⊎ AST) ▶ AST
‘ASTC’ = ! expr ⊗ ‘ExprC’
```

```
        ⊕ ! decl ⊗ 'DeclC'
'ASTF' : ⊤ ▶ AST
'ASTF' = Fix 'ASTC'
```

In 'ASTC' (the code for the family before closing the fixed point) we encode either an expression or a declaration, each coupled with an equality proof that forces the output index to match the datatype we are defining. If we now select the expr index from the interpretation of 'ASTF', then ! decl yields an uninhabited type, whereas ! expr yields a trivial equality, thereby ensuring that only 'ExprF' corresponds to the structure in this case. If we select decl in turn, then only 'DeclF' contributes to the structure.

We can now define the conversion functions between the original datatypes and the representation:

```
mutual
  toExpr : {r : ⊤ → Set} → ⟦ 'ASTF' ⟧ r expr → Expr (r tt)
  toExpr ⟨ inj₁ (refl , inj₁ x) ⟩                  = econst x
  toExpr ⟨ inj₁ (refl , inj₂ (inj₁ (x , y))) ⟩     = add (toExpr x) (toExpr y)
  toExpr ⟨ inj₁ (refl , inj₂ (inj₂ (inj₁ x))) ⟩    = evar x
  toExpr ⟨ inj₁ (refl , inj₂ (inj₂ (inj₂ (d , e)))) ⟩ = elet (toDecl d) (toExpr e)
  toExpr ⟨ inj₂ (() , _) ⟩

  toDecl : {r : ⊤ → Set} → ⟦ 'ASTF' ⟧ r decl → Decl (r tt)
  toDecl ⟨ inj₁ (() , _) ⟩
  toDecl ⟨ inj₂ (refl , inj₁ (x , e)) ⟩   = assign x (toExpr e)
  toDecl ⟨ inj₂ (refl , inj₂ (d₁ , d₂)) ⟩ = seq (toDecl d₁) (toDecl d₂)
```

The important difference from the previous examples is that now we have absurd patterns. For instance, in toExpr we have to produce an Expr, so the generic value starting with inj_2 is impossible, since there is no inhabitant of the type expr ≡ decl. Dually, in the conversion from the original type into the generic type, these proofs have to be supplied:

```
mutual
  fromExpr : {r : ⊤ → Set} → Expr (r tt) → ⟦ 'ASTF' ⟧ r expr
  fromExpr (econst x) = ⟨ inj₁ (refl , inj₁ x) ⟩
  fromExpr (add x y)  = ⟨ inj₁ (refl , inj₂ (inj₁ (fromExpr x , fromExpr y))) ⟩
  fromExpr (evar x)   = ⟨ inj₁ (refl , inj₂ (inj₂ (inj₁ x))) ⟩
  fromExpr (elet d e) = ⟨ inj₁ (refl , inj₂ (inj₂ (inj₂ (fromDecl d , fromExpr e)))) ⟩

  fromDecl : {r : ⊤ → Set} → Decl (r tt) → ⟦ 'ASTF' ⟧ r decl
  fromDecl (assign x e) = ⟨ inj₂ (refl , inj₁ (x , fromExpr e)) ⟩
  fromDecl (seq d₁ d₂)  = ⟨ inj₂ (refl , inj₂ (fromDecl d₁ , fromDecl d₂)) ⟩
```

At this stage the proofs are trivial to produce (refl), since we know exactly the type of the index.

6.1.8 Arbitrarily indexed datatypes

The index types of a functor need not be finite types. Consider the following datatype describing lists of a fixed length (vectors):

```
infixr 5 _::_
data Vec (A : Set) : ℕ → Set where
  []  :                          Vec A ze
  _::_ : {n : ℕ} → A → Vec A n → Vec A (su n)
```

The type Vec is indexed by the type of natural numbers ℕ. In fact, for a given type A, Vec A defines a family of sets: Vec A ze (which contains only the empty list), Vec A (su ze) (all possible singleton lists), and so on. As such, we can see Vec as a code with one input parameter (the type A) and ℕ output parameters:

```
'VecC' : ℕ → (⊤ ⊎ ℕ) ▶ ℕ
'VecC' ze      = ! ze
'VecC' (su n)  = ! (su n) ⊗ I (inj₁ tt) ⊗ I (inj₂ n)
```

Note, however, that we need to parameterise 'VecC' by a natural number, since the code depends on the particular value of the index. In particular, a vector of length su n is an element together with a vector of length n. Unlike in the AST example, we cannot simply sum up the codes for all the different choices of output index, because there are infinitely many of them. Therefore, we need yet another code in our universe:

```
Σ : {C : ⊥ ▶ ⊤} → (⟦ C ⟧ (const ⊤) tt → I ▶ O) → I ▶ O
```

The code Σ introduces an existential datatype:

```
data ∃ {A : Set} (B : A → Set) : Set where
  some : ∀ {x} → B x → ∃ B
⟦ Σ f ⟧ r o = ∃ (λ i → ⟦ f i ⟧ r o)
```

Note that Σ is parametrised by a function f that takes values to codes. Arguments of f are supposed to be indices, but we would like them to be described by codes again, since that makes it easier to define generic functions over the universe. Therefore, we make a compromise and choose a code for a single unparametrised datatype ⊥ ▶ ⊤ rather than an arbitrary Set—for more discussion, see Section 6.1.10.

To create a value of type ⟦ Σ f ⟧ we need a specific witness i to obtain a code from f. When using a value of type ⟦ Σ f ⟧ we can access the index stored in the existential.

Here is the map function for Σ:

```
map (Σ g) f o (some {i} x)  =  some (map (g i) f o x)
```

Using Σ, we can finalise the encoding of Vec:

```
'VecF' : ⊤ ▶ ℕ
'VecF' = Fix (Σ {C = 'ℕ'} 'VecC')
```

We make use of the fact that we already have a code 'ℕ' for our index type of natural numbers.

Finally we can provide the conversion functions. We need to pattern-match on the implicit natural number to be able to provide it as existential evidence in fromVec', and to be able to produce the right constructor in toVec':

```
fromVec : ∀ {n r} → Vec (r tt) n → ⟦ 'VecF' ⟧ r n
fromVec {n = ze}   []       = ⟨ some {x = ze}   refl ⟩
fromVec {n = su m} (h :: t) = ⟨ some {x = su m} (refl , (h , fromVec t)) ⟩
toVec : ∀ {n r} → ⟦ 'VecF' ⟧ r n → Vec (r tt) n
toVec ⟨ some {ze}   refl ⟩                = []
toVec ⟨ some {su n} (refl , (h , t)) ⟩ = h :: toVec t
```

6.1.9 Nested datatypes

Nested datatypes [Bird and Meertens, 1998] can be encoded in Agda using indexed datatypes. Consider the type of perfectly balanced binary trees:

```
data Perfect (A : Set) : {n : ℕ} → Set where
  split : {n : ℕ} → Perfect A {n} × Perfect A {n} → Perfect A {su n}
  leaf  : A → Perfect A {ze}
```

Perfect trees are indexed over the naturals. A perfect tree is either a leaf, which has depth ze, or a split-node, which has depth su n and contains two subtrees of depth n each.

In Haskell, this type is typically encoded by changing the parameters of the type in the return type of the constructors: split would have return type Perfect (Pair A), for some suitable Pair type. We can define Perfect' as such a nested datatype in Agda, too:

```
data Pair (A : Set) : Set where
  pair : A → A → Pair A
data Perfect' : Set → Set₁ where
  split : {A : Set} → Perfect' (Pair A) → Perfect' A
  leaf  : {A : Set} → A                 → Perfect' A
```

Now, Perfect' A is isomorphic to a dependent pair of a natural number n and an element of Perfect A {n}.

We can therefore reduce the problem of encoding Perfect' to the problem of encoding Perfect, which in turn can be done similarly to the encoding of vectors shown in the previous section:

```
'PerfectC' : ℕ → (⊤ ⊎ ℕ) ▶ ℕ
'PerfectC' (ze)   = ! ze      ⊗ I (inj₁ tt)
'PerfectC' (su n) = ! (su n) ⊗ I (inj₂ n) ⊗ I (inj₂ n)
```

```
'PerfectF' : ⊤ ▶ ℕ
'PerfectF' = Fix (Σ {C = 'ℕ'} 'PerfectC')
```

We omit the embedding-projection pair as it provides no new insights.

6.1.10 Summary and discussion

At this point we are at the end of showing how various sorts of datatypes can be encoded in the `indexed` universe. For reference, we now show the entire universe, its interpretation, and the map function:

```
data _▶_ (I : Set) (O : Set) : Set₁ where
  Z      : I ▶ O
  U      : I ▶ O
  I      : I → I ▶ O
  !      : O → I ▶ O
  _⊕_    :          I ▶ O → I ▶ O → I ▶ O
  _⊗_    :          I ▶ O → I ▶ O → I ▶ O
  _◎_    : ∀ {M} → M ▶ O → I ▶ M → I ▶ O
  Fix    : (I ⊎ O) ▶ O → I ▶ O
  _↗_↘_  : ∀ {I' O'} → I' ▶ O' → (I' → I) → (O → O') → I ▶ O
  Σ      : {C : ⊥ ▶ ⊤} → (⟦ C ⟧ (const ⊤) tt → I ▶ O) → I ▶ O
  Iso    : (C : I ▶ O) → (D : I ▷ O) →
           ((r : Indexed I) → (o : O) → D r o ≃ ⟦ C ⟧ r o) → I ▶ O

data μ {I O : Set} (F : (I ⊎ O) ▶ O) (r : Indexed I) (o : O) : Set where
  ⟨_⟩ : ⟦ F ⟧ (r | μ F r) o → μ F r o

⟦_⟧ : ∀ {I O} → I ▶ O → I ▷ O
⟦ Z         ⟧ r o = ⊥
⟦ U         ⟧ r o = ⊤
⟦ I i       ⟧ r o = r i
⟦ F ↗ f ↘ g ⟧ r o = ⟦ F ⟧ (r ∘ f) (g o)
⟦ F ⊕ G     ⟧ r o = ⟦ F ⟧ r o ⊎ ⟦ G ⟧ r o
⟦ F ⊗ G     ⟧ r o = ⟦ F ⟧ r o × ⟦ G ⟧ r o
⟦ F ◎ G     ⟧ r o = ⟦ F ⟧ (⟦ G ⟧ r) o
⟦ Fix F     ⟧ r o = μ F r o
⟦ ! o'      ⟧ r o = o ≡ o'
⟦ Σ f       ⟧ r o = ∃ (λ i → ⟦ f i ⟧ r o)
⟦ Iso C D e ⟧ r o = D r o

map : {I O : Set} {r s : Indexed I}
      (C : I ▶ O) → (r ⇉ s) → (⟦ C ⟧ r ⇉ ⟦ C ⟧ s)
```

```
map Z           f o ()
map U           f o x                      =  x
map (I i)       f o x                      =  f i x
map (F ⊕ G)     f o (inj₁ x)               =  inj₁ (map F f o x)
map (F ⊕ G)     f o (inj₂ x)               =  inj₂ (map G f o x)
map (F ⊗ G)     f o (x , y)                =  map F f o x , map G f o y
map (F ↗ g ↘ h) f o x                      =  map F (f ∘ g) (h o) x
map (F ⊙ G)     f o x                      =  map F (map G f) o x
map (! o')      f o x                      =  x
map (Σ g)       f o (some {i} x)           =  some (map (g i) f o x)
map (Fix F)     f o ⟨ x ⟩                  =  ⟨ map F (f ‖ map (Fix F) f) o x ⟩
map {r = r} {s = s} (Iso C D e) f o x with (e r o , e s o)
... | (ep₁ , ep₂)                          =  to ep₂ (map C f o (from ep₁ x))
                                              where open _≃_
```

Naturally, there are several variations possible in this approach, and the universe we have shown is not the only useful spot in the design space. We will now briefly discuss a number of choices we have taken.

Perhaps most notably, we have avoided the inclusion of arbitrary constants of the form

```
K : Set → I ▶ O
```

There are two main reasons why we might want to have constants in universes. One is to refer to user-defined datatypes; we can do this via Iso, as long as the user-defined datatypes can be isomorphically represented by a code. The other reason is to be able to include abstract base types (say, floating point numbers), for which it is difficult to give a structural representation.

Adding constants, however, introduces problems as well. While map is trivial to define for constants—they are just ignored—most other functions, such as e.g. decidable equality, become impossible to define in the presence of arbitrary constants. Additional assumptions (such as that the constants being used admit decidable equality themselves) must usually be made, and it is impossible to predict in advance all the constraints necessary when defining a K code.

A similar problem guides our rather pragmatic choice when defining Σ. There are at least two other potential definitions for Σ:

```
Σ₁ : Set → I ▶ O
Σ₂ : ∀ {I' O' r' o'} {C : I' ▶ O'} → (⟦ C ⟧ r' o' → I' ▶ O') → I ▶ O
```

In the first case we allow an arbitrary Set as index type. This, however, leads to problems with decidable equality [Morris, 2007, Section 3.3], because in order to compare two existential pairs for equality we have to compare the indices. Restricting the indices to representable types guarantees we can easily compare them. The second variant is more general than our current Σ, abstracting from a code with any input and output indices. However, this makes the interpretation depend on additional parameters r' and o', which

we are unable to produce, in general. In our Σ we avoid this problem by setting I' to ⊥ and O' to ⊤, so that r' is trivially λ () and o' is tt. Our Σ encodes indexing over a single unparametrised datatype.

Note that the previous approaches we have seen can be obtained from `indexed` by instantiating the input and output indices appropriately. The `regular` library considers functors defining single datatypes without parameters. This corresponds to instantiating the input index to ⊥ ⊎ ⊤ (no parameters, one recursive slot) and the output index to ⊤, and allowing fixed points only on the outside, of type ⊥ ⊎ ⊤ ▶ ⊤ → ⊥ ▶ ⊤.

Adding one parameter brings us to the realm of `polyp`, with fixed points of bifunctors. The kind of the fixed-point operator in `polyp` corresponds exactly to the type ⊤ ⊎ ⊤ ▶ ⊤ → ⊤ ▶ ⊤, i.e. taking a bifunctor to a functor. Note also that `polyp` furthermore allows a limited form of composition where the left operand is a fixed point of a bifunctor (i.e. of type ⊤ ▶ ⊤), and the right operand is a bifunctor (of type ⊤ ⊎ ⊤ ▶ ⊤).

Finally, `multirec` supports any finite number n of mutually recursive datatypes without parameters; that corresponds to fixed points of type ⊥ ⊎ Fin n ▶ Fin n → ⊥ ▶ Fin n.

The `indexed` approach thus generalises all of the previous libraries. It allows arbitrarily many mutually-recursive datatypes with arbitrarily many parameters, and it allows non-finite index types. We formalise this claim in Chapter 8.

In the remainder of this chapter we provide further evidence of the usefulness of the `indexed` universe by showing a number of applications.

6.2 Functor laws

To be able to state the functor laws for the functorial map of Section 6.1.4 we must first present some auxiliary definitions for composition and equality of index-preserving mappings:

```
_ ∘⇉ _ : {I : Set} {r s t : Indexed I} → s ⇉ t → r ⇉ s → r ⇉ t
(f ∘⇉ g) i x = f i (g i x)

infix 4 _ ≡⇉ _
_ ≡⇉ _ : ∀ {I : Set} {r s : Indexed I} (f g : r ⇉ s) → Set
f ≡⇉ g = ∀ i x → f i x ≡ g i x
```

We cannot use the standard propositional equality directly in our laws because Agda's propositional equality on functions amounts to intensional equality, and we cannot prove our functions to be intensionally equal without postulating an extensionality axiom.

We can now state the functor laws, which in the setting of indexed sets and index-preserving functions take the following form:

```
mapid : {I O : Set} {r : Indexed I} (C : I ▶ O) →
        map {r = r} C id⇉ ≡⇉ id⇉
map∘ : {I O : Set} {r s t : Indexed I} (C : I ▶ O) (f : s ⇉ t) (g : r ⇉ s) →
        map C (f ∘⇉ g) ≡⇉ (map C f) ∘⇉ (map C g)
```

Recall that $\mathsf{id}_{\rightrightarrows}$ is the identity mapping, introduced in Section 6.1.5.

The first step to prove the laws is to define a congruence lemma stating that mapping equivalent functions results in equivalent maps:

```
map∀ : {I O : Set} {r s : Indexed I} {f g : r ⇉ s}
       (C : I ▶ O) → f ≡⇉ g → map C f ≡⇉ map C g
map∀ Z          ip i ()
map∀ U          ip i x = refl
map∀ (I i')     ip i x = ip i' x
map∀ (F ↗ f ↘ g) ip i x = map∀ F (ip ∘ f) (g i) x
```

For `I` we use the proof of equality of the functions. For re-indexing we need to adapt the indices appropriately. Sums, products, and composition proceed recursively and rely on congruence of the constructors:

```
map∀ (F ⊕ G) ip i (inj₁ x) = cong inj₁ (map∀ F ip i x)
map∀ (F ⊕ G) ip i (inj₂ x) = cong inj₂ (map∀ G ip i x)
map∀ (F ⊗ G) ip i (x , y)  = cong₂ _,_ (map∀ F ip i x) (map∀ G ip i y)
map∀ (F ◎ G) ip i x        = map∀ F (map∀ G ip) i x
map∀ (! o')     ip i x     = refl
```

For `Σ`, fixed points, and isomorphisms, the proof also proceeds recursively and by re-sorting to congruence of equality where necessary:

```
map∀ (Σ g)   ip i (some x) = cong some (map∀ (g _) ip i x)
map∀ (Fix F) ip i ⟨ x ⟩    = cong ⟨_⟩ (map∀ F (||-cong ip (map∀ (Fix F) ip)) i x)
map∀ {r = r} {s = s} (Iso C D e) ip i x =
  cong (to (e s i)) (map∀ C ip i (from (e r i) x)) where open _≃_
```

Note the use of an underscore in an expression in the case for `Σ`; this just means that Agda can automatically infer the only possible argument for that position. We also open the `_≃_` record so that we can use its operations unqualified.

For fixed points we use a lemma regarding congruence of the `_||_` operator:

```
||-cong : {I J : Set} {r u : Indexed I} {s v : Indexed J}
          {f₁ f₂ : r ⇉ u} {g₁ g₂ : s ⇉ v} →
          f₁ ≡⇉ f₂ → g₁ ≡⇉ g₂ → f₁ || g₁ ≡⇉ f₂ || g₂
||-cong if ig (inj₁ i) x = if i x
||-cong if ig (inj₂ i) x = ig i x
```

We are now able to prove the functor laws. We start with $\mathsf{map}^{\mathsf{id}}$:

```
mapid : {I O : Set} {r : Indexed I} (C : I ▶ O) →
        map {r = r} C id⇉ ≡⇉ id⇉
```

```
map^id Z           i ()
map^id U           i x  =  refl
map^id (I i')      i x  =  refl
map^id (F ↗ f ↘ g) i x  =  map^id F (g i) x
```

The cases for Z, U, and I are trivial. For re-indexing we convert the output index and continue recursively; the input index is converted implicitly by Agda, filling in the implicit parameter r with r ∘ f. Sums and products proceed recursively with congruence, while composition requires transitivity as well:

```
map^id (F ⊕ G) i (inj₁ x)  =  cong inj₁ (map^id F i x)
map^id (F ⊕ G) i (inj₂ x)  =  cong inj₂ (map^id G i x)
map^id (F ⊗ G) i (x , y)   =  cong₂ _,_ (map^id F i x) (map^id G i y)
map^id (F ⊙ G) i x         =  trans (map^∀ F (map^id G) i x) (map^id F i x)
```

Tagging obeys the law trivially, and for Σ we proceed recursively with congruence on the some constructor:

```
map^id (! o') i x          =  refl
map^id (Σ g) i (some x)    =  cong some (map^id (g _) i x)
```

The more complicated cases are those for fixed points and isomorphisms. For fixed points we first need an auxiliary lemma for equality between identities over the _||_ operator, similar to ||-cong for congruence:

```
||-id : {I J : Set} {r : Indexed I} {s : Indexed J} {f : r ⇉ r} {g : s ⇉ s} →
        f ≡⇉ id⇉ → g ≡⇉ id⇉ → (f || g) ≡⇉ id⇉
||-id if ig (inj₁ i) x  =  if i x
||-id if ig (inj₂ i) x  =  ig i x
```

With this lemma we can prove the identity law for a fixed point. Recall that fixed points have left- and right-tagged codes. The map function proceeds recursively on the right (recursive occurrences), and it applies the mapping function directly on the left (parameters). So we have to show that both components are the identity transformation. We use the ||-id lemma for this, with a trivial proof for the left and a recursive call for the right:

```
map^id (Fix F) i ⟨ x ⟩  =  cong ⟨_⟩ (trans (map^∀ F (||-id (λ _ _ → refl)
                                                        (map^id (Fix F))) i x)
                                           (map^id F i x))
```

We are left with isomorphisms. The proof requires only a careful use of symmetry and transitivity of propositional equality:

```
map^id {r = r} (Iso C D e) i x  =  sym (trans (sym ((iso₁ (e r i)) x))
                                              (sym (cong (to (e r i)))
```

```
                                    (map^id C i (from (e r i) x)))))
  where open _≃_
```

Direct uses of sym and trans usually result in proof terms that are hard to write and understand; writing our proofs in equational reasoning style would be preferable. Fortunately, the standard library provides a module with convenient syntax for writing proofs in equational reasoning style:

```
map^id {r = r} (Iso C D e) i x =
  begin
    to (e r i) (map C id⇉ i (from (e r i) x))
  ≡⟨ cong (to (e r i)) (map^id C i (from (e r i) x)) ⟩
    to (e r i) (from (e r i) x)
  ≡⟨ iso₁ (e r i) x ⟩
    x ∎
  where open _≃_
```

This style makes it clear that the proof consists of two steps: removing the map C id⇉ application via a recursive call to the proof (of type map C id⇉ ≡⇉ id⇉), and using one of the isomorphism proofs. See the work of Mu et al. [2009] for a detailed account on this style of proofs in Agda.

We have seen how to prove the identity functor law for the `indexed` map. The composition law map° is similar, so we omit it.

6.3 Generic decidable equality

Generic functions can be defined by instantiating standard recursion patterns such as the catamorphism defined in Section 6.1.5, or directly, as a type-indexed computation by pattern-matching on the universe codes. Here we show how to define a semi-decidable equality for our universe; in case the compared elements are equal, we return a proof of the equality. In case the elements are different we could return a proof of their difference; however, for conciseness, we only show the proof of equality.

The type of decidable equality is:

```
deqt : {I O : Set} {r : Indexed I} (C : I ▶ O) → SemiDec r → SemiDec (⟦ C ⟧ r)
```

Note that to compute the equality SemiDec (⟦ C ⟧ r) we need the equality on the respective recursive indexed functors (SemiDec r). We use a type constructor SemiDec to define our type of semi-decidable equality:

```
SemiDec : ∀ {I} → Indexed I → Set
SemiDec r = ∀ i → (x y : r i) → Maybe (x ≡ y)
```

That is, for all possible indices, given two indexed functors we either return a proof of their equality or fail.

The type constructor _⋔_ is a variation on _||_ (Section 6.1.4) adapted to the type of SemiDec, necessary for the Fix alternative:

```
infixr 5 _⋔_
_⋔_ : {I J : Set} {r : Indexed I} {s : Indexed J} →
      SemiDec r → SemiDec s → SemiDec (r | s)
(f ⋔ g) (inj₁ i) = f i
(f ⋔ g) (inj₂ i) = g i
```

Decidable equality is impossible on Z, trivial on U, and dispatches to the supplied function f on the selected index i for I:

```
deqt Z     f o () y
deqt U     f o tt tt = just refl
deqt (I i) f o x  y  = f i x y
```

Re-indexing proceeds recursively on the arguments, after adjusting the recursive equality and changing the output index:

```
deqt (F ↗ f ↘ g) h o x y = deqt F (h ∘ f) (g o) x y
```

Sums are only equal when both components have the same constructor. In that case, we recursively compute the equality, and apply congruence to the resulting proof with the respective constructor:

```
deqt (F ⊕ G) f o (inj₁ x) (inj₂ y) = nothing
deqt (F ⊕ G) f o (inj₂ x) (inj₁ y) = nothing
deqt (F ⊕ G) f o (inj₁ x) (inj₁ y) = deqt F f o x y >>= just ∘ cong inj₁
deqt (F ⊕ G) f o (inj₂ x) (inj₂ y) = deqt G f o x y >>= just ∘ cong inj₂
```

We use _ >>= _ : {A B : Set} → Maybe A → (A → Maybe B) → Maybe B as the monadic bind to combine Maybe operations. The product case follows similarly, using a congruence lifted to two arguments:

```
deqt (F ⊗ G) f o (x₁ , x₂) (y₁ , y₂) =      deqt F f o x₁ y₁ >>=
                                       λ l → deqt G f o x₂ y₂ >>=
                                       λ r → just (cong₂ _,_ l r)
```

A composition F ⊙ G represents a code F containing Gs at the recursive positions. Equality on this composition is the equality on F using the equality on G for the recursive positions. Tagging is trivial after pattern-matching:

```
deqt (F ⊙ G) f o  x    y    = deqt F (deqt G f) o x y
deqt (! o)     f .o refl refl = just refl
```

Note the .o pattern, meaning that the second o is necessarily the same as the first one, due to equality constraints introduced by the pattern-match on refl.

For Σ, we first check if the witnesses (the first components of the dependent pair) are equal. We make essential use of the fact that the type of the witnesses is representable within the universe, so we can reuse the decidable equality function we are just defining. If the witnesses are equal, we proceed to compare the elements, using the code produced by the index, and apply congruence with some to the resulting proof:

```
deqt (Σ {C = C} g) f o (some {i₁} x) (some {i₂} y)
  with deqt {r = λ _ → _} C (λ ()) tt i₁ i₂
deqt (Σ g) f o (some {i} x) (some y) | nothing  =  nothing
deqt (Σ g) f o (some {i} x) (some y) | just refl  =  deqt (g i) f o x y
                                                  >>=  just ∘ cong some
```

Equality for fixed points uses the auxiliary operator _⋔_ introduced before to apply f to parameters and deqt to recursive calls:

```
deqt (Fix F) f o ⟨ x ⟩ ⟨ y ⟩  =  deqt F (f ⋔ deqt (Fix F) f) o x y  >>=  just ∘ cong ⟨_⟩
```

Finally, equality for Iso uses the proof of equivalence of the isomorphism. We omit that case here as it is lengthy and unsurprising.

We are now ready to test our decidable equality function. We start by instantiating it to natural numbers:

```
deqtℕ : (m n : ℕ) → Maybe (m ≡ n)
deqtℕ = deqt {r = (λ ())} 'ℕ' (λ ()) tt
```

The type of natural numbers has no input indices, therefore we supply the absurd function λ () as the argument for the equality on the inputs. The function works as expected:

```
deqtExample₁ : deqtℕ (su ze) (su ze) ≡ just refl
deqtExample₁ = refl

deqtExample₂ : deqtℕ (su ze) ze ≡ nothing
deqtExample₂ = refl
```

For lists of naturals, we supply the deqtℕ function we have just defined as the argument for the equality on input indices:

```
deqtList : {A : Set} →
           ((x y : A) → Maybe (x ≡ y)) → (l₁ l₂ : [ A ]) → Maybe (l₁ ≡ l₂)
deqtList {A} f x y = deqt {r = const A} 'List' (const f) tt x y
```

We can now define some example lists for testing:

```
l₁ : [ ℕ ]
l₁ = ze :: su ze :: []
```

```
l₂ : [ ℕ ]
l₂ = ze :: ze :: []
```

And confirm that equality works as expected:

```
deqtExample₃ : deqtList deqtℕ l₁ l₁ ≡ just refl
deqtExample₃ = refl
deqtExample₄ : deqtList deqtℕ l₁ l₂ ≡ nothing
deqtExample₄ = refl
```

6.4 A zipper for indexed functors

Along with defining generic functions, we can also define type-indexed datatypes [Hinze et al., 2002]: types defined generically in the universe of representations. A frequent example is the type of one-hole contexts, described for regular types by McBride [2001], and for `multirec` by Rodriguez Yakushev et al. [2009], with application to the zipper [Huet, 1997].

We revisit the zipper in the context of the `indexed` universe, starting with the type-indexed datatype of one-hole contexts, and proceeding to the navigation functions.

6.4.1 Generic contexts

The one-hole context of an indexed functor is the type of values where exactly one input position is replaced by a hole. The idea is that we can then split an indexed functor into an input value and its context. Later, we can identify a position in a datatype by keeping a stack of one-hole contexts that gives us a path from the subtree in focus up to the root of the entire structure.

We define Ctx as another interpretation function for our universe: it takes a code and the input index indicating what kind of position we want to replace by a hole, and it returns an indexed functor:

```
Ctx : {I O : Set} → I ▶ O → I → I ▷ O
Ctx Z     i r o = ⊥
Ctx U     i r o = ⊥
Ctx (! o') i r o = ⊥
Ctx (I i') i r o = i ≡ i'
```

For the void, unit, and tag types there are no possible holes, so the context is the empty datatype. For I, we have a hole if and only if the index for the hole matches the index we recurse on. If there is a hole, we want the context to be isomorphic to the unit type, otherwise it should be isomorphic to the empty type. An equality type of i and i' has the desired property.

For a re-indexed code we store proofs of the existence of the new indices together with the reindexed context:

```
Ctx (F ↗ f ↘ g) i r o = ∃ (λ i' → ∃ (λ o' → f i' ≡ i × g o ≡ o' × Ctx F i' (r ∘ f) o'))
```

As described by McBride [2001], computing the one-hole context of a polynomial functor corresponds to computing the formal derivative. This correspondence motivates the definitions for sum, product, and composition. Notably, the context for a composition follows the chain rule:

```
Ctx (F ⊕ G) i r o = Ctx F i r o ⊎ Ctx G i r o
Ctx (F ⊗ G) i r o = Ctx F i r o × ⟦ G ⟧ r o ⊎ ⟦ F ⟧ r o × Ctx G i r o
Ctx (F ⊙ G) i r o = ∃ (λ m → Ctx F m (⟦ G ⟧ r) o × Ctx G i r m)
```

The context of a Σ f is the context of the resulting code f i', for some appropriate index i'. The context of an Iso is the context of the inner code:

```
Ctx (Σ f)       i r o = ∃ (λ i' → Ctx (f i') i r o)
Ctx (Iso C D e) i r o = Ctx C i r o
```

The context of a fixed point is more intricate. Previous zippers (such as that of `multirec`) have not directly considered the context for a fixed point, since these were outside the universe. If we are interested in positions of an index i within a structure that is a fixed point, we must keep in mind that there can be many such positions, and they can be located deep down in the recursive structure. A Fix F is a layered tree of F structures. When we finally find an i, we must therefore be able to give an F-context for i for the layer in which the input actually occurs. Now F actually has more inputs than Fix F, so the original index i corresponds to the index inj_1 i for F. We then need a path from the layer where the hole is back to the top. To store this path, we define a datatype of context-stacks Ctxs. This stack consists of yet more F-contexts, but each of the holes in these F-contexts must correspond to a recursive occurrence, i.e. an index marked by inj_2:

```
Ctx (Fix F) i r o = ∃ (λ j → Ctx F (inj₁ i) (r ∣ μ F r) j × Ctxs F j r o)

data Ctxs {I O : Set} (F : (I ⊎ O) ▶ O) (i : O) (r : Indexed I) : Indexed O where
  empty : Ctxs F i r i
  push  : {j o : O} → Ctx F (inj₂ i) (r ∣ μ F r) j → Ctxs F j r o → Ctxs F i r o
```

Note that the stack of contexts Ctxs keeps track of two output indices, just like a single context Ctx. A value of type Ctxs F i r o denotes a stack of contexts for a code F, focused on a hole with type index i, on an expression of type index o. This stack is later reused in the higher-level navigation functions (Section 6.4.4).

6.4.2 Plugging holes in contexts

A basic operation on contexts is to replace the hole by a value of the correct type; this is called "plugging". Its type is unsurprising: given a code C, a context on C with hole

of type index i, and a value of this same index, plug returns the plugged value as an interpretation of C:

```
plug : {I O : Set} {r : Indexed I} {i : I} {o : O} →
       (C : I ▶ O) → Ctx C i r o → r i → ⟦ C ⟧ r o
```

Plugging is not defined for the codes with an empty context type. For I, pattern-matching on the context gives us a proof that the value to plug has the right type, so we return it:

```
plug Z     ()   r
plug U     ()   r
plug (! o) ()   r
plug (I i) refl r = r
```

Re-indexing proceeds plugging recursively, after matching the equality proofs:

```
plug (F ↗ f ↘ g) (some (some (refl , refl , c))) r = plug F c r
```

Plugging on a sum proceeds recursively on the alternatives. Plugging on a product has two alternatives, depending on whether the hole lies on the first or on the second component. Plugging on a composition F ⊙ G proceeds as follows: we obtain two contexts, an F-context c with a G-shaped hole, and a G-context d with a hole of the type that we want to plug in. So we plug r into d, and the resulting G is then plugged into c:

```
plug (F ⊕ G) (inj₁ c)       r = inj₁ (plug F c r)
plug (F ⊕ G) (inj₂ c)       r = inj₂ (plug G c r)
plug (F ⊗ G) (inj₁ (c , g)) r = plug F c r , g
plug (F ⊗ G) (inj₂ (f , c)) r = f           , plug G c r
plug (F ⊙ G) (some (c , d)) r = plug F c (plug G d r)
```

Plugging into a fixed-point structure is somewhat similar to the case of composition, only that instead of two layers, we now deal with an arbitrary number of layers given by the stack. We plug our r into the first context c, and then unwind the stack using an auxiliary function unw. Once the stack is empty we are at the top and done. Otherwise, we proceed recursively upwards, plugging each level as we go:

```
plug {r = s} {o = o} (Fix F) (some {m} (c , cs)) r = unw m cs ⟨ plug F c r ⟩
  where unw : ∀ m → Ctxs F m s o → ⟦ Fix F ⟧ s m → ⟦ Fix F ⟧ s o
        unw .o empty          x = x
        unw m (push {o} c cs) x = unw o cs ⟨ plug F c x ⟩
```

Finally, plugging on Σ proceeds recursively, using the code associated with the index packed in the context. For isomorphisms we proceed recursively on the new code and apply the to conversion function to the resulting value:

```
plug (Σ f) (some {i} c) r = some (plug (f i) c r)
plug {r = s} {o = o} (Iso C D e) x r with e s o
... | ep = to ep (plug C x r) where open _≃_
```

6.4.3 Primitive navigation functions: first and next

With plug we can basically move up in the zipper: after plugging a hole we are left with a value of the parent type. To move down, we need to be able to split a value into its first child and the rest. This is the task of first:

```
first : {A I O : Set} {r : Indexed I} {o : O} → (C : I ▶ O) →
        ((i : I) → r i → Ctx C i r o → Maybe A) → ⟦ C ⟧ r o → Maybe A
```

We write first in continuation-passing style. One should read it as a function taking a value and returning a context with a hole at the first (i.e. leftmost) possible position, the value previously at that position, and its index. These are the three arguments to the continuation function. Since not all values have children, we might not be able to return a new context, so we wrap the result in a Maybe. Note that first (and not the caller) picks the index of the hole according to the first input that it can find.

As there are no values of void type, this case is impossible. For unit and tag types, there are no elements, so the split fails:

```
first Z     k ()
first U     k x  =  nothing
first (! o) k x  =  nothing
```

For I there is exactly one child, which we return by invoking the continuation:

```
first (I i) k x  =  k i x refl
```

For re-indexing and sums we proceed recursively, after adapting the continuation function to the new indices and context appropriately:

```
first (F ↗ f ↘ g) k x  =  first F (λ i' r c → k (f i') r (some (some (refl , (refl , c))))) x
first (F ⊕ G) k (inj₁ x)  =  first F (λ i r c → k i r (inj₁ c)) x
first (F ⊕ G) k (inj₂ x)  =  first G (λ i r c → k i r (inj₂ c)) x
```

On a product we have a choice. We first try the first component, and only in case of failure (through plusMaybe) we try the second:

```
first (F ⊗ G) k (x , y)  =  plusMaybe (first F (λ i r c → k i r (inj₁ (c , y))) x)
                                      (first G (λ i r c → k i r (inj₂ (x , c))) y)
```

Composition follows the nested structure: we first split the outer structure, and if that is successful, we call first again on the obtained inner structure:

```
first (F ⊙ G) k x  =  first F (λ m s c → first G (λ i r d → k i r (some (c , d))) s) x
```

Fixed points require more care. We use two mutually-recursive functions to handle the possibility of having to navigate deeper into recursive structures until we find an element, building a stack of contexts as we go. Note that the type of input indices

changes once we go inside a fixed point. If we obtain a split on an inj_1-marked value, then that is an input index of the outer structure, so we are done. If we get a split on an inj_2-marked value, we have hit a recursive occurrence. We then descend into that by calling fstFix again, and retain the current layer on the context stack:

```
first {A} {I} {O} {r} {o} (Fix F) k x = fstFix x empty where
  mutual
    fstFix : {m : O} → μ F r m → Ctxs F m r o → Maybe A
    fstFix ⟨ x ⟩ cs = first F (contFix cs) x
    contFix : {m : O} → Ctxs F m r o → (i : I ⊎ O) →
              (r | μ F r) i → Ctx F i (r | μ F r) m → Maybe A
    contFix cs (inj₁ i) r c = k i r (some (c , cs))
    contFix cs (inj₂ i) r c = fstFix r (push c cs)
```

Splitting on a Σ proceeds recursively as usual, and on isomorphisms we apply conversion functions where necessary:

```
first (Σ f) k (some {i'} y) = first (f i') (λ i r c → k i r (some c)) y
first {r = r} {o = o} (Iso C D e) k x with e r o
... | ep = first C k (from ep x) where open _≃_
```

Another primitive navigation function is next, which, given a current context and an element which fits in the context, tries to move the context to the next element to the right, producing a new context and an element of a compatible (and possibly different) type:

```
next : {A I O : Set} {r : Indexed I} {o : O} → (C : I ▶ O) →
       ((i : I) → r i → Ctx C i r o → Maybe A) →
       {i : I} → Ctx C i r o → r i → Maybe A
```

Its implementation is similar to that of first, so we omit it.

6.4.4 Derived navigation

Given the primitives plug, first, and next, we are ready to define high-level navigation functions, entirely hiding the context from the user. Recursive user datatypes are generally defined as a top-level application of Fix to another code. We thus define a zipper data structure that enables the user to navigate through a structure defined by a fixed point on the outside. We can then efficiently navigate to all the recursive positions in that structure. Note that variations of this approach are possible, such as a zipper that also allows navigating to parameter positions, or defining navigation functions that operate on an Iso code.

While we are traversing a structure, we keep the current state in a datatype that we call a *location*. It contains the subtree that is currently in focus, and a path up to the root of the complete tree. The path is a stack of one-hole contexts, and we reuse the Ctxs type from Section 6.4.1 to hold the stack:

```
data Loc {I O : Set} (F : (I ⊎ O) ▶ O) (r : Indexed I) (o : O) : Set where
  loc : {o' : O} → ⟦ Fix F ⟧ r o' → Ctxs F o' r o → Loc F r o
```

The high-level navigation functions all have the same type:

```
Nav : Set₁
Nav = ∀ {I O} {F : (I ⊎ O) ▶ O} {r : Indexed I} {o : O} →
      Loc F r o → Maybe (Loc F r o)
```

Given a location, we might be able to move to a new location, keeping the same code, interpretation for recursive positions, and output type index. We need to allow for failure since, for instance, it is not possible to move down when there are no children.

Moving down corresponds to splitting the context using first:

```
down : Nav
down {F = F} (loc ⟨ x ⟩ cs) = first F (λ { (inj₁ i) r d → nothing
                                         ; (inj₂ i) r d → just (loc r (push d cs)) }) x
```

Here we can use Agda's syntax for pattern matching on anonymous lambdas when defining the continuation function. This function expresses the behavior for the different kinds of input positions: we do not descend into parameters, and on recursive calls we build a new location by returning the tree in focus and pushing the new layer on the stack.

Moving up corresponds to plugging in the current context. It fails if there is no context, meaning we are already at the root:

```
up : Nav
up           (loc x empty)      = nothing
up {F = F} (loc x (push c cs)) = just (loc (⟨ plug F c x ⟩) cs)
```

Moving to the right corresponds to getting the next child. We process the results of next as in down, this time using an auxiliary definition for the continuation:

```
right : Nav
right                          (loc x empty)          = nothing
right {I} {O} {F} {r} {o} (loc x (push {m} c cs)) = next F auxf c x
  where auxf : (i : I ⊎ O) → (r ∣ μ F r) i →
               Ctx F i (r ∣ μ F r) m → Maybe (Loc F r o)
        auxf (inj₁ i) r d = nothing
        auxf (inj₂ i) r d = just (loc r (push d cs))
```

The functions presented so far allow reaching every position in the datatype. Other navigation functions, such as to move left, can be added in a similar way.

Finally, we provide operations to start and stop navigating a structure:

```
enter : ∀ {I O} {F : (I ⊎ O) ▶ O} {r : Indexed I} {o : O} →
        ⟦ Fix F ⟧ r o → Loc F r o
```

```
enter x  =  loc x empty
leave  :  ∀ {I O} {F  :  (I ⊎ O) ▶ O} {r  :  Indexed I} {o  :  O}  →
          Loc F r o  →  Maybe ([[ Fix F ]] r o)
leave (loc x empty)      =  just x
leave (loc x (push h t))  =  up (loc x (push h t))  >>=  leave
```

To enter we create a location with an empty context, and to leave we move up until the context is empty.

It is also useful to be able to manipulate the focus of the zipper. The function update applies a type-preserving function to the current focus:

```
update  :  ∀ {I O} {F  :  (I ⊎ O) ▶ O} {r  :  Indexed I}  →
           [[ Fix F ]] r ⇉ [[ Fix F ]] r  →  Loc F r ⇉ Loc F r
update f _ (loc x l)  =  loc (f _ x) l
```

6.4.5 Examples

We now show how to use the zipper on the Rose datatype of Section 6.1.6. The representation type of rose trees is non-trivial since it uses composition with lists (and therefore contains an internal fixed point). Let us define an example tree:

```
treeB  :  Rose ℕ  →  Rose ℕ
treeB t  =  fork 5
              (fork 4 [] :: (fork 3 [] :: (fork 2 [] :: (fork 1
                                                          (t :: []) :: []))))
tree  :  Rose ℕ
tree  =  treeB (fork 0 [])
```

Our example tree has a node 5 with children numbered 4 through 1. The last child has one child of its own, labelled 0.

We now define a function that navigates through this tree by entering, going down (into the child labelled 4), moving right three times (to get to the rightmost child), descending down once more (to reach the child labelled 0), and finally increments this label:

```
navTree  :  Rose ℕ  →  Maybe (Rose ℕ)
navTree t  =  down (enter (fromRose t))
          >>=  right  >>=  right  >>=  right  >>=  down
          >>=  just ∘ update (const (map 'Rose' (const su) tt)) _
          >>=  leave  >>=  just ∘ toRose
```

Since the navigation functions return Maybe, but other functions (e.g. enter and fromRose) do not, combining these functions requires care. However, it would be easy to define a small combinator language to overcome this problem and simplify writing traversals with the zipper.

We can check that our traversal behaves as expected:

```
navTreeExample : navTree tree ≡ just (treeB (fork 1 []))
navTreeExample = refl
```

We can also use the zipper on the Perfect nested datatype of Section 6.1.9, for instance. In this example, we enter a perfect tree, move down into the first (thus leftmost) child, swap all the elements of this subtree, and then leave:

```
swapLeft : {n : ℕ} → Perfect ℕ {n} → Maybe (Perfect ℕ {n})
swapLeft p = down (enter (fromPerfect p))
           >>= just ∘ (update f _)
           >>= leave
           >>= just ∘ toPerfect
  where f : (n : ℕ) → ⟦'Perfect' ⟧ (T ℕ) n → ⟦'Perfect' ⟧ (T ℕ) n
        f n p = fromPerfect (cataPerfect {R = λ n → Perfect ℕ {n}} leaf
                                         (λ a → split a ∘ swap) (toPerfect p))
```

For swapping we use a catamorphism with the function cataPerfect, which is just a specialisation of cata to the type of perfect trees. We can check that swapLeft behaves as expected:

```
p₀ : Perfect ℕ
p₀ = split 0 (leaf , leaf)

p₁ : Perfect ℕ
p₁ = split 1 (leaf , leaf)

p : Perfect ℕ → Perfect ℕ → Perfect ℕ
p x y = split 6 (split 2 (x , y) , split 5 (p₀ , p₁))

swapLeftExample : swapLeft (p p₀ p₁) ≡ just (p p₁ p₀)
swapLeftExample = refl
```

We have seen how to define a zipper for the universe of `indexed`. In particular, we have seen a type of one-hole contexts for fixed points, using a stack of contexts that is normally used only for the higher-level navigation functions. Even though this zipper is more complex than that of `multirec`,[1] it operates through all the codes in this universe, meaning that we can now zip through indexed datatypes, for instance.

6.5 Conclusion and future work

In this chapter we have seen a universe of indexed functors for dependently typed generic programming in Agda which is both intuitive, in the sense that its codes map naturally to datatype features, and powerful, since it supports a wide range of datatypes and allows defining a wide range of datatype-generic behavior.

[1] `http://hackage.haskell.org/package/zipper`

The key features of the `indexed` approach are: support for parametrised datatypes and recursive positions in a uniform way, fixed points as part of the universe, support for general composition, and the possibility to incorporate isomorphisms between datatypes into the universe. These features make it possible to reuse codes once defined, and to make normal user-defined Agda datatypes available for use in generic programs.

Along the way we have seen how proofs of correctness easily integrate with indexed functors, both in the universe and in the generic functions defined. Furthermore, we have shown a generalisation of the zipper operation to the `indexed` universe, allowing for efficient and type-safe generic traversals.

Options for future work include generalising from the zipper to dissection [McBride, 2008] as well as refining the universe further, for example by allowing generalisation over arity and datatype kind, both present in the work of Weirich and Casinghino [2010]. Furthermore, we lack a mechanism for automatic generation of datatype representations, and we cannot precisely state which datatypes are not representable in our universe.

The original goal that inspired the `indexed` approach was to overcome the limitations of `multirec`, which can handle mutually recursive families of datatypes, but does not support parametrised datatypes or composition. While `indexed` overcomes these limitations, it is only implemented in Agda. Agda clearly has many advantages for generic programming, but Haskell is currently still superior when it comes to writing practical code. We hope that the newly introduced kind-level features in GHC 7.4 [Yorgey et al., 2012] will allow us to obtain a reasonably elegant encoding of `indexed` in Haskell, bringing the power of this approach to a wider audience.

CHAPTER 7

The `instant-generics` library

`instant-generics` is another approach to generic programming with type families, initially described by Chakravarty et al. [2009]. It distinguishes itself from the other approaches we have discussed so far in that it does not represent recursion via a fixed-point combinator. Like `regular`, `instant-generics` has also been used to implement a generic rewriting library [Van Noort et al., 2010]. To allow meta-variables in rewrite rules to occur at any position (i.e. not only in recursive positions), type-safe runtime casts are performed to determine if the type of the meta-variable matches that of the expression. The generic programming support recently built into the Glasgow and Utrecht Haskell compilers, which we discuss in Chapter 11, is partly inspired by `instant-generics`, and shares a number of features.

7.1 Agda model

In the original encoding of `instant-generics` by Chakravarty et al. in Haskell, recursive datatypes are handled through indirect recursion between the conversion functions (from and to) and the generic functions. This is a form of a shallow encoding, but different from the one we have seen in Section 3.2; in `regular` we know the type of the recursive elements, so we carry around a function to apply at the recursive positions. `instant-generics`, on the other hand, relies on Haskell's class system to recursively apply a generic function at the recursive points of a datatype.

We find that the most natural way to model this in Agda is to use coinduction [Danielsson and Altenkirch, 2010]. This allows us to define infinite codes, and generic functions operating on these codes, while still passing the termination check. This encoding would also be appropriate for other Haskell approaches without a fixed-point operator, such as

"Generics for the Masses" [Hinze, 2006] and "Lightweight Implementation of Generics and Dynamics" [Cheney and Hinze, 2002]. Although approaches without a fixed-point operator have trouble expressing recursive morphisms, they have been popular in Haskell because they easily allow encoding datatypes with irregular forms of recursion (such as mutually recursive or nested [Bird and Meertens, 1998] datatypes).

Coinduction in Agda is built around the following three operations:

```
∞ : ∀ ( A : Set) → Set
♯ : ∀ {A : Set} → A → ∞ A
♭ : ∀ {A : Set} → ∞ A → A
```

The ∞ operator is used to label coinductive occurrences. A type ∞ A can be seen as a delayed computation of type A, with associated delay (♯) and force (♭) functions. Since Agda is a total language, all computations are required to terminate; in particular, values are required to be finite. Coinduction lifts this restriction by allowing the definition of coinductive types which need not be finite, only productive. Productivity is checked by the termination checker, which requires corecursive definitions to be guarded by coinductive constructors. The universe of codes for `instant-generics` has a code R for tagging coinductive occurrences of codes:

```
data Code : Set₁ where
  U   :                      Code
  K   : Set                → Code
  R   : (C : ∞ Code)       → Code
  _⊕_ : (C D : Code)       → Code
  _⊗_ : (C D : Code)       → Code
```

Compared to the previous approaches, the code K for arbitrary Sets is also a novelty. We have not introduced a code for constants in any of the Agda models so far because its inclusion is trivial and unsurprising. For `instant-generics`, however, it becomes necessary for the embeddings in Chapter 8.

We give the interpretation as a datatype to ensure that it is inductive. The judicious use of the coinduction primitive ♭ makes the Agda encoding pass the termination checker, as the definitions remain productive:

```
data ⟦_⟧ : Code → Set₁ where
  tt   :                                          ⟦ U     ⟧
  k    : {A : Set}       → A                    → ⟦ K A   ⟧
  rec  : {C : ∞ Code}    → ⟦ ♭ C ⟧              → ⟦ R C   ⟧
  inj₁ : {C D : Code}    → ⟦ C ⟧                → ⟦ C ⊕ D ⟧
  inj₂ : {C D : Code}    → ⟦ D ⟧                → ⟦ C ⊕ D ⟧
  _,_  : {C D : Code}    → ⟦ C ⟧ → ⟦ D ⟧        → ⟦ C ⊗ D ⟧
```

As an example, consider the encoding of lists in `instant-generics`:

```
‘List’ : Set → Code
‘List’ A = U ⊕ (K A ⊗ R (♯ ‘List’ A))
```

Although the definition of ‘List’ is directly recursive, it is accepted by the termination checker, since it remains productive.

Due to the lack of fixed points, we cannot write a map function on the recursive positions. But we can easily write other recursive generic functions, such as a traversal that crushes a term into a result of a fixed return type:

```
crush : {R : Set}
        (C : Code) → (R → R → R) → (R → R) → R → ⟦ C ⟧ → R
crush U       _⊞_ ⇑ 𝟙 _       = 𝟙
crush (K _)   _⊞_ ⇑ 𝟙 _       = 𝟙
crush (R C)   _⊞_ ⇑ 𝟙 (rec x) = ⇑ (crush (♭ C) _⊞_ ⇑ 𝟙 x)
crush (C ⊕ D) _⊞_ ⇑ 𝟙 (inj₁ x) = crush C _⊞_ ⇑ 𝟙 x
crush (C ⊕ D) _⊞_ ⇑ 𝟙 (inj₂ x) = crush D _⊞_ ⇑ 𝟙 x
crush (C ⊗ D) _⊞_ ⇑ 𝟙 (x , y) = (crush C _⊞_ ⇑ 𝟙 x) ⊞ (crush D _⊞_ ⇑ 𝟙 y)
```

Function crush is similar to map in the sense that it can be used to define many generic functions. It takes three arguments that specify how to combine pairs of results (_⊞_), how to adapt the result of a recursive call (⇑), and what to return for constants and constructors with no arguments (𝟙).[1] However, crush is unable to change the type of datatype parameters, since `instant-generics` has no knowledge of parameters.

We can compute the size of a structure as a crush, for instance:

```
size : (C : Code) → ⟦ C ⟧ → ℕ
size C = crush C _+_ su ze
```

Here we combine multiple results by adding them, increment the total at every recursive call, and ignore constants and units for size purposes. We can test that this function behaves as expected on lists:

```
aList : ⟦ ‘List’ ⊤ ⟧
aList = inj₂ (k tt , rec (inj₂ (k tt , rec (inj₁ tt))))

testSize : size _ aList ≡ 2
testSize = refl
```

While a map function cannot be defined like in the previous approaches, traversal and transformation functions can still be expressed in `instant-generics`. In particular, if we are willing to exchange static by dynamic type checking, type-safe runtime casts can be performed to compare the types of elements being mapped against the type expected by the mapping function, resulting in convenient to use generic functions [Van Noort et al., 2010]. However, runtime casting is known to result in poor runtime performance, as it prevents the compiler from performing type-directed optimisations (as those of Chapter 9).

[1] It is worth noting that crush is itself a simplified catamorphism for the Code type.

7.2 Haskell implementation

The `instant-generics` library is encoded in Haskell with the following representation types:

```
data U       = U
data Var α   = Var α
data Rec α   = Rec α
data α :+: β = L α | R β
data α :×: β = α :×: β
```

The main difference from the previous approaches is that we no longer need to carry additional parameters to encode recursion. Every argument to a constructor is wrapped in a Var or Rec tag to indicate if the argument is a parameter of the type or a (potentially recursive) occurrence of a datatype. These correspond to the codes K and R in our Agda model, respectively.

7.2.1 Datatype representation

As customary, we use a type class to mediate between a value and its generic representation:

```
class Representable α where
  type Rep α
  to   :: Rep α → α
  from :: α     → Rep α
```

A Representable type has an associated Representation type, which is constructed using the types shown before. We use a type family to encode the isomorphism between a type and its representation, together with conversion functions to and from.

As an example, we show the instantiation of the standard list datatype:

```
instance Representable [α] where
  type Rep [α] = U :+: (Var α :×: Rec [α])
  from []                 = L U
  from (h : t)            = R (Var h :×: Rec t)
  to (L U)                = []
  to (R (Var h :×: Rec t)) = h : t
```

Lists have two constructors: the empty case corresponds to a Left injection, and the cons case to a Right injection. For the cons case we have two parameters, encoded using a product. The first one is a Variable, and the second a Recursive occurrence of the list type.

7.2.2 Generic functions

Generic functions are defined by giving an instance for each representation type. To define the crush function we have seen in the Agda model we first define a type class:

```
class CrushRep α where
  crushRep :: (ρ → ρ → ρ) → (ρ → ρ) → ρ → α → ρ
```

We call this class CrushRep because we only give instances for the representation types. The instances for units, sums, and products are unsurprising:

```
instance CrushRep U where
  crushRep _ _ z _ = z
instance (CrushRep α, CrushRep β) ⇒ CrushRep (α :+: β) where
  crushRep f g z (L x) = crushRep f g z x
  crushRep f g z (R x) = crushRep f g z x
instance (CrushRep α, CrushRep β) ⇒ CrushRep (α :×: β) where
  crushRep f g z (x :×: y) = f (crushRep f g z x) (crushRep f g z y)
```

For variables we also stop the traversal and return the unit result:

```
instance CrushRep (Var α) where
  crushRep f g z (Var x) = z
```

For recursive types, however, we have to recursively apply crushRep. In the previous approaches we carried around a function to apply at the recursive indices. In `instant-generics` we rely on indirect recursion:

```
instance (Crush α) ⇒ CrushRep (Rec α) where
  crushRep f g z (Rec x) = g (crush f g z x)
```

We use another class, Crush, to define how to crush user datatypes. Since the representation in `instant-generics` is shallow, at the Rec positions we have user types, not representation types. So we recursively invoke the crush function, which performs the same task as crushRep, but on user datatypes.

The user-facing Crush class is defined as follows:

```
class Crush α where
  crush :: (ρ → ρ → ρ) → (ρ → ρ) → ρ → α → ρ
```

We also define a function crushDefault that is suitable for using when defining generic instances of Crush:

```
crushDefault :: (Representable α, CrushRep (Rep α)) ⇒
                (ρ → ρ → ρ) → (ρ → ρ) → ρ → α → ρ
crushDefault f g z = crushRep f g z ∘ from
```

Instantiating this generic function to representable datatypes is now very simple:

```
instance Crush [ α ] where
  crush = crushDefault
```

To finish the same example as in the Agda model, we define a function to compute the size of a term as a crush:

```
size :: (Crush α) ⇒ α → Int
size = crush (+) (+1) 0
```

Now we can confirm that size (1 : 2 : []) evaluates to 2.

We have seen a library for generic programming that does not use a fixed-point view on data. This gives us more flexibility when encoding datatypes, but we lose the ability to express certain generic behaviour, such as recursive morphisms.

CHAPTER 8

Comparing the approaches

We have seen five different libraries for generic programming: `regular` (Chapter 3), `polyp` (Chapter 4), `multirec` (Chapter 5), `indexed` (Chapter 6), and `instant-generics` (Chapter 7). We have modelled each of these libraries in Agda by reducing the approach to its core elements (universe and its interpretation). We have also seen example datatype encodings and generic functions. In this chapter we relate the approaches to each other, formally, by providing conversion functions between approaches and proofs of correctness.

8.1 Introduction

The abundance of generic programming approaches is not a new problem. Including pre-processors, template-based approaches, language extensions, and libraries, there are well over 15 different approaches to generic programming in Haskell [Brown and Sampson, 2009, Chakravarty et al., 2009, Cheney and Hinze, 2002, Hinze, 2006, Hinze and Löh, 2006, Hinze and Peyton Jones, 2001, Hinze et al., 2006, Jansson and Jeuring, 1997, Kiselyov, 2006, Lämmel and Peyton Jones, 2003, 2004, 2005, Löh, 2004, Magalhães et al., 2010a, Mitchell and Runciman, 2007, Norell and Jansson, 2004, Oliveira et al., 2006, Rodriguez Yakushev et al., 2009, Weirich, 2006]. This abundance is caused by the lack of a clearly superior approach; each approach has its strengths and weaknesses, uses different implementation mechanisms, a different generic view [Holdermans et al., 2006] (i.e. a different structural representation of datatypes), or focuses on solving a particular task. Their number and variety makes comparisons difficult, and can make prospective generic programming users struggle even before actually writing a generic program, since first they have to choose a library that is adequate to their needs.

Some effort has been made in comparing different approaches to generic programming from a practical point of view [Hinze et al., 2007, Rodriguez Yakushev et al., 2008], or to classify approaches [Hinze and Löh, 2009]. While Generic Haskell [Löh, 2004] has been formalised in different attempts [Verbruggen et al., 2010, Weirich and Casinghino, 2010], no formal comparison between modern approaches has been attempted, leaving a gap in the knowledge of the relationships between each approach. We argue that this gap should be filled; for starters, a formal comparison provides a theoretical foundation for understanding different generic programming approaches and their differences and similarities. However, the contribution is not merely of theoretical interest, since a comparison can also provide means for converting between approaches. Ironically, code duplication across generic programming libraries is evident: the same function can be nearly identical in different approaches, yet impossible to reuse, due to the underlying differences in representation. With a formal proof of inclusion between two approaches a conversion function can be derived, removing the duplication of generic code, as we show in Section 8.3.

In this chapter we take the initial steps towards a formal comparison of generic programming approaches. We establish relationships among the five Agda encodings shown before, and reason about them. While the inclusion relations are the ones we expect, the way to convert between approaches is often far from straightforward, and reveals subtle differences between the approaches. Each inclusion is evidenced by a conversion function that brings codes from one universe into another, enabling generic function reuse across different approaches.

Our proofs are fully machine-checked (in Agda), but written in equational reasoning style, so that they resemble handwritten proofs, and remain clear and elegant. A few caveats remain, namely with regard to termination and universe levels inconsistency, which we discuss in more detail in Section 8.4.

8.2 Embedding relation

In this section we show which approaches can be *embedded* in other approaches. When we say that approach `A` embeds into approach `B`, we mean that the interpretation of any code defined in approach `A` has an equivalent interpretation in approach `B`. The starting point of an embedding is a code-conversion function that maps codes from approach `A` into approach `B`.

For this comparison we consider the Agda models for the approaches presented before. The `indexed` approach has been shown in full detail, while the other four approaches have been stripped down to their core. Therefore we consider only a subset of `indexed` for this section. The minimised `indexed` universe we use in this section follows:

```
data Codeᵢ (I O : Set) : Set₁ where
  U    :                          Codeᵢ I O
  I    : I                   → Codeᵢ I O
  !    : O                   → Codeᵢ I O
  _⊕_  : (F G : Codeᵢ I O) → Codeᵢ I O
```

```
_⊗_ : (F G : Codeᵢ I O) → Codeᵢ I O
_◎_ : {M : Set} → (F : Codeᵢ M O) → (G : Codeᵢ I M) → Codeᵢ I O
Fix : (F : Codeᵢ (I ⊎ O) O) → Codeᵢ I O
```

Recall that we used an infix type operator _►_ to represent the universe of `indexed`; for consistency, we name that operator Code_i in this section. We elide Z as this code does not add much expressiveness to the universe, and is not present in the other approaches. Isomorphisms (Iso) are also removed for simplicity, as they do not add expressive power, only convenience. The codes for Σ and reindexing (_↗__↘_), however, add considerable power to the `indexed` universe. Yet we decide to remove them for comparative purposes so that we can relate `indexed` to `instant-generics`.[1]

Figure 8.1 presents a graphical view of the embedding relation between the five approaches; the arrows mean "embeds into". Note that the embedding relation is naturally transitive. As expected, `multirec` and `polyp` both subsume `regular`, but they don't subsume each other, since one supports families of recursive types and the other supports one parameter. They are however both subsumed by `indexed`. Finally, the liberal encoding of `instant-generics` allows encoding at least all the types supported by the other approaches (even if it doesn't support the same operations on those types, such as catamorphisms).

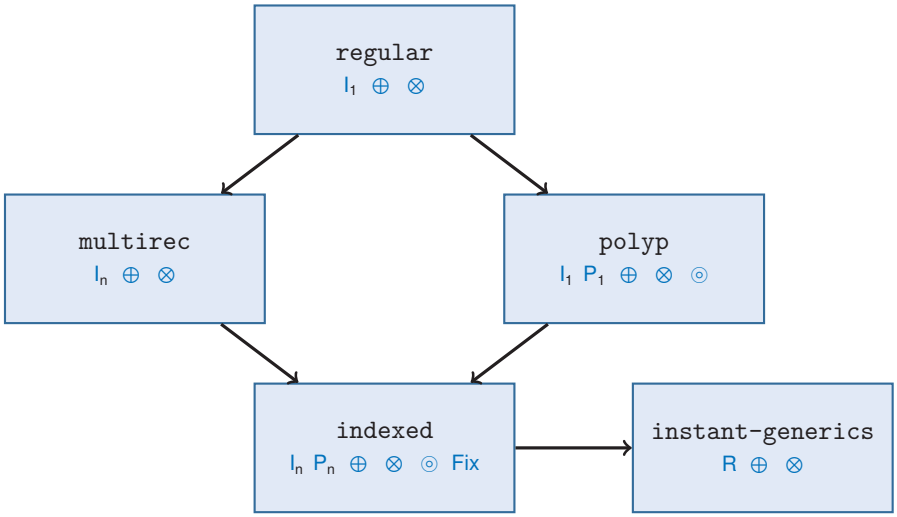


Figure 8.1: Embedding relation between the approaches

We now discuss each of the five conversions and their proofs.

[1] The minimised `indexed` shown in this section identifies a subset of full `indexed` that is expressive enough to generalise `regular`, `polyp`, and `multirec`, while remaining embedabble into `instant-generics`.

8.2.1 Regular to PolyP

We start with the simplest relation: embedding `regular` into `polyp`. The first step is to convert `regular` codes into `polyp` codes:

```
⇑r→p : Coder → Codep
⇑r→p Ur          = Up
⇑r→p Ir          = Ip
⇑r→p (F ⊕r G)    = (⇑r→p F) ⊕p (⇑r→p G)
⇑r→p (F ⊗r G)    = (⇑r→p F) ⊗p (⇑r→p G)
```

Since all libraries share similar names, we use subscripts to denote the library we are referring to; `r` for `regular` and `p` for `polyp`. All `regular` codes embed trivially into `polyp`, so $\Uparrow_{r\to p}$ is unsurprising. After having defined the code conversion we can show that the interpretation of a code in `regular` is equivalent to the interpretation of the converted code in `polyp`. We do this by defining an isomorphism (see Section 2.2.3) between the two interpretations (and their fixed points). We show one direction of the conversion, from `regular` to `polyp`:

```
from r→p : {α : Set} (C : Coder) → ⟦ C ⟧r α → ⟦ ⇑r→p C ⟧p ⊥ α
from r→p Ur        tt       = tt
from r→p Ir        x        = x
from r→p (F ⊕r G) (inj1 x)  = inj1 (from r→p F x)
from r→p (F ⊕r G) (inj2 x)  = inj2 (from r→p G x)
from r→p (F ⊗r G) (x , y)   = from r→p F x , from r→p G y
from μr→p : (C : Coder) → μr C → μp (⇑r→p C) ⊥
from μr→p C ⟨ x ⟩r = ⟨ from r→p C (mapr C (from μr→p C) x) ⟩p
```

Since `regular` does not support parameters, we set the `polyp` parameter to $\bot$ for `regular` codes. Function $\text{from}_{r\to p}$ does the conversion of one layer, while $\text{from}\mu_{r\to p}$ ties the recursive knot by expanding fixed points, converting one layer, and mapping itself recursively. Unfortunately $\text{from}\mu_{r\to p}$ (and indeed all conversions in this section involving fixed points) does not pass Agda's termination checker; we provide some insights on how to address this problem in Section 8.4.

The conversion in the other direction ($\text{to}_{r\to p}$ and $\text{to}\mu_{r\to p}$) is entirely symmetrical:

```
to r→p : {α : Set} (C : Coder) → ⟦ ⇑r→p C ⟧p ⊥ α → ⟦ C ⟧r α
to r→p Ur        tt       = tt
to r→p Ir        x        = x
to r→p (F ⊕r G) (inj1 x)  = inj1 (to r→p F x)
to r→p (F ⊕r G) (inj2 x)  = inj2 (to r→p G x)
to r→p (F ⊗r G) (x , y)   = to r→p F x , to r→p G y
to μr→p : (C : Coder) → μp (⇑r→p C) ⊥ → μr C
to μr→p C ⟨ x ⟩p = ⟨ to r→p C (mapp (⇑r→p C) id (to μr→p C) x) ⟩r
```

Having defined the conversion functions, we now have to prove that they indeed form an isomorphism. We first consider the case without fixed points:

```
iso1r→p : {R : Set} →
          (C : Coder) → (x : ⟦ C ⟧r R) → tor→p C (fromr→p C x) ≡ x
iso1r→p Ur        _       = refl
iso1r→p Ir        _       = refl
iso1r→p (F ⊕r G) (inj1 x) = cong inj1 (iso1r→p F x)
iso1r→p (F ⊕r G) (inj2 x) = cong inj2 (iso1r→p G x)
iso1r→p (F ⊗r G) (x , y)  = cong2 _,_ (iso1r→p F x) (iso1r→p G y)
```

This proof is trivial, and so is the proof for $\mathsf{from}_{r\to p}\ C\ (\mathsf{to}_{r\to p}\ C\ x) \equiv x$, its counterpart for the other direction.

When considering fixed points the proofs become more involved, since recursion has to be taken into account. However, using the equational reasoning module from the standard library (see the work of Mu et al. [2009] for a detailed account on this style of proofs in Agda) we can keep the proofs readable:

```
open ≡-Reasoning
isoμ1r→p : (C : Coder) (x : μr C) → toμr→p C (fromμr→p C x) ≡ x
isoμ1r→p C ⟨ x ⟩r = cong ⟨_⟩r $
  begin
    tor→p C (mapp (⇑r→p C) id (toμr→p C) (fromr→p C (mapr C (fromμr→p C) x)))
  ≡⟨ mapCommute^p_r C _ ⟩
    mapr C (toμr→p C) (tor→p C (fromr→p C (mapr C (fromμr→p C) x)))
  ≡⟨ cong (mapr C (toμr→p C)) (iso1r→p C _) ⟩
    mapr C (toμr→p C) (mapr C (fromμr→p C) x)
  ≡⟨ map^∘_r C ⟩
    mapr C (toμr→p C ∘ fromμr→p C) x
  ≡⟨ map^∀_r C (isoμ1r→p C) x ⟩
    mapr C id x
  ≡⟨ map^id_r C ⟩
    x ∎
```

In this proof we start with an argument relating the maps of `regular` and `polyp`:

```
mapCommute^p_r : {α β : Set} {f : α → β} (C : Coder) (x : ⟦ ⇑r→p C ⟧p ⊥ α) →
                 tor→p C (mapp (⇑r→p C) id f x) ≡ mapr C f (tor→p C x)
```

In words, this theorem states that the following two operations over a `regular` term `x` that has been lifted into `polyp` are equivalent:

- To map a function f over the recursive positions of x in `polyp` with map_p and then convert to `regular`;
- To first convert x back into `regular`, and then map the function f with map_r.

After this step, we either proceed by a recursive argument (referring to $\mathsf{iso}_{1_{\mathsf{r}\to\mathsf{p}}}$ or $\mathsf{iso}\mu_{1_{\mathsf{r}\to\mathsf{p}}}$) or by a lemma. For conciseness, we show only the types of the lemmas:

$$\begin{aligned}
&\mathsf{map}_\mathsf{r}^\circ \;:\; \{\alpha\ \beta\ \gamma : \mathsf{Set}\}\ \{f : \beta \to \gamma\}\ \{g : \alpha \to \beta\}\ (C : \mathsf{Code}_\mathsf{r})\\
&\qquad \{x : [\![\, C \,]\!]_\mathsf{r}\ \alpha\} \to \mathsf{map}_\mathsf{r}\ C\ f\ (\mathsf{map}_\mathsf{r}\ C\ g\ x) \equiv \mathsf{map}_\mathsf{r}\ C\ (f \circ g)\ x\\
&\mathsf{map}_\mathsf{r}^\forall \;:\; \{\alpha\ \beta : \mathsf{Set}\}\ \{f\ g : \alpha \to \beta\}\ (C : \mathsf{Code}_\mathsf{r}) \to\\
&\qquad (\forall\ x \to f\ x \equiv g\ x) \to (\forall\ x \to \mathsf{map}_\mathsf{r}\ C\ f\ x \equiv \mathsf{map}_\mathsf{r}\ C\ g\ x)\\
&\mathsf{map}_\mathsf{r}^\mathsf{id} \;:\; \{\alpha : \mathsf{Set}\}\ (C : \mathsf{Code}_\mathsf{r})\ \{x : [\![\, C \,]\!]_\mathsf{r}\ \alpha\} \to \mathsf{map}_\mathsf{r}\ C\ \mathsf{id}\ x \equiv x
\end{aligned}$$

These lemmas are standard properties of map_r, namely the functor laws and the fact that map_r preserves extensional equality (a form of congruence on map_r). All of them are easily proven by induction on the codes, similarly to the proofs for the `indexed` map of Section 6.2.

Put together, $\mathsf{from}\mu_{\mathsf{r}\to\mathsf{p}}$, $\mathsf{to}\mu_{\mathsf{r}\to\mathsf{p}}$, $\mathsf{iso}\mu_{1_{\mathsf{r}\to\mathsf{p}}}$, and $\mathsf{iso}\mu_{2_{\mathsf{r}\to\mathsf{p}}}$ (the dual of $\mathsf{iso}\mu_{1_{\mathsf{r}\to\mathsf{p}}}$) form an isomorphism that shows how to embed `regular` codes into `polyp` codes:

$$\begin{aligned}
&\mathsf{Regular}{\simeq}\mathsf{PolyP} \;:\; (C : \mathsf{Code}_\mathsf{r}) \to \mu_\mathsf{r}\ C \simeq \mu_\mathsf{p}\ (\Uparrow_{\mathsf{r}\to\mathsf{p}}\ C)\ \bot\\
&\mathsf{Regular}{\simeq}\mathsf{PolyP}\ C \;=\; \mathbf{record}\ \{\mathsf{from} \;=\; \mathsf{from}\mu_{\mathsf{r}\to\mathsf{p}}\ C\\
&\qquad\qquad\qquad\qquad\quad ;\mathsf{to} \;=\; \mathsf{to}\mu_{\mathsf{r}\to\mathsf{p}}\ \ C\\
&\qquad\qquad\qquad\qquad\quad ;\mathsf{iso}_1 \;=\; \mathsf{iso}\mu_{1_{\mathsf{r}\to\mathsf{p}}}\ C\\
&\qquad\qquad\qquad\qquad\quad ;\mathsf{iso}_2 \;=\; \mathsf{iso}\mu_{2_{\mathsf{r}\to\mathsf{p}}}\ C\}
\end{aligned}$$

8.2.2 Regular to Multirec

The conversion from `regular` to `multirec` (subscript m) is very similar to the conversion into `polyp`. We convert a `regular` code into a `multirec` code with a unitary index type, since `regular` codes define a single type:

$$\begin{aligned}
&\Uparrow_{\mathsf{r}\to\mathsf{m}} \;:\; \mathsf{Code}_\mathsf{r} \to \mathsf{Code}_\mathsf{m}\ \top\\
&\Uparrow_{\mathsf{r}\to\mathsf{m}}\ \mathsf{U}_\mathsf{r} \;=\; \mathsf{U}_\mathsf{m}\\
&\Uparrow_{\mathsf{r}\to\mathsf{m}}\ \mathsf{I}_\mathsf{r} \;=\; \mathsf{I}_\mathsf{m}\ \mathsf{tt}\\
&\Uparrow_{\mathsf{r}\to\mathsf{m}}\ (F \oplus_\mathsf{r} G) \;=\; (\Uparrow_{\mathsf{r}\to\mathsf{m}}\ F) \oplus_\mathsf{m} (\Uparrow_{\mathsf{r}\to\mathsf{m}}\ G)\\
&\Uparrow_{\mathsf{r}\to\mathsf{m}}\ (F \otimes_\mathsf{r} G) \;=\; (\Uparrow_{\mathsf{r}\to\mathsf{m}}\ F) \otimes_\mathsf{m} (\Uparrow_{\mathsf{r}\to\mathsf{m}}\ G)
\end{aligned}$$

Instead of defining conversion functions from and to, we can directly express the equality between the interpretations:

$$\begin{aligned}
&\uparrow_{\mathsf{r}\to\mathsf{m}} \;:\; \{\alpha : \mathsf{Set}\} \to (C : \mathsf{Code}_\mathsf{r}) \to [\![\, C \,]\!]_\mathsf{r}\ \alpha \equiv [\![\ \Uparrow_{\mathsf{r}\to\mathsf{m}}\ C\ ]\!]_\mathsf{m}\ (\lambda\ _ \to \alpha)\ \mathsf{tt}\\
&\uparrow_{\mathsf{r}\to\mathsf{m}}\ \mathsf{U}_\mathsf{r} \;=\; \mathsf{refl}\\
&\uparrow_{\mathsf{r}\to\mathsf{m}}\ \mathsf{I}_\mathsf{r} \;=\; \mathsf{refl}
\end{aligned}$$

```
↑r→m (F ⊕r G)  =  cong₂ _⊎_ (↑r→m F) (↑r→m G)
↑r→m (F ⊗r G)  =  cong₂ _×_ (↑r→m F) (↑r→m G)
```

We elide the isomorphism between the fixed points as it is similar to that for `polyp`.

8.2.3 PolyP to Indexed

We proceed to the conversion between `polyp` and `indexed` (subscript i) codes. As we mentioned before, particular care has to be taken with composition; the remaining codes are trivially converted, so we only show composition:

```
⇑p→i : Codep → Codei (⊤ ⊎ ⊤) ⊤
⇑p→i (F ⊙p G)  =  (Fixi (⇑p→i F)) ⊙i (⇑p→i G)
```

We cannot simply take the `indexed` composition of the two converted codes because their types do not allow composition. A `polyp` code results in an open `indexed` code with one parameter and one recursive position, therefore of type $\mathsf{Code_i}\ (\top \uplus \top)\ \top$. Taking the fixed point of such a code gives a code of type $\mathsf{Code_i}\ \top\ \top$, so we can mimic `polyp`'s interpretation of composition in our conversion, using the $\mathsf{Fix_i}$ of `indexed`. In fact, converting `polyp` to `indexed` helps us understand the interpretation of composition in `polyp`, because the types now show us that there is no way to define a composition other than by combining it with the fixed-point operator.

Converting composed values from `polyp` is then a recursive task, due to the presence of fixed points. We first convert the outer functor F, and then map the conversion onto the arguments, recalling that on the left we have parameter codes G, while on the right we have recursive occurrences of the original composition:

```
from⇑p→i : {α ρ : Set} (C : Codep) →
           ⟦ C ⟧p α ρ → ⟦ ⇑p→i C ⟧i ((const α) |i (const ρ)) tt
from⇑p→i (F ⊙p G) ⟨ x ⟩p  =
  ⟨ mapi (⇑p→i F) ((λ _ → from⇑p→i G) ||i (λ _ → from⇑p→i (F ⊙p G))) tt
      (from⇑p→i F x) ⟩i
```

We also show the conversion in the opposite direction, which is entirely symmetrical:

```
to⇑p→i : {α ρ : Set}
         (C : Codep) → ⟦ ⇑p→i C ⟧i ((const α) |i (const ρ)) tt → ⟦ C ⟧p α ρ
to⇑p→i (F ⊙p G) ⟨ x ⟩i  =
  ⟨ to⇑p→i F (mapi (⇑p→i F) ((λ _ → to⇑p→i G) ||i (λ _ → to⇑p→i (F ⊙p G))) tt x) ⟩p
```

The conversion of `polyp` fixed points ignores parameters and maps itself recursively over recursive positions:

```
fromμp→i : {α : Set} (C : Codep) → μp C α → ⟦ Fixi (⇑p→i C) ⟧i (const α) tt
fromμp→i C ⟨ x ⟩p  =  ⟨ from⇑p→i C (mapp C id (fromμp→i C) x) ⟩i
```

We omit the conversion in the opposite direction for brevity, and also the isomorphism proof. Both the case for composition and for `polyp` fixed points require the properties of map_i from Section 6.2.

8.2.4 Multirec to Indexed

The conversion from `multirec` to `indexed` is simpler than the conversion from `polyp` into `indexed` because `multirec` does not support composition. We embed a `multirec` family indexed over a set I into an `indexed` family with output index I, and input index $\bot \uplus$ I (no parameters, same number of outputs, before taking the fixed point):

$$
\begin{aligned}
&\Uparrow_{m\to i} : \{I : \mathsf{Set}\} \to \mathsf{Code}_m\ I \to \mathsf{Code}_i\ (\bot \uplus I)\ I\\
&\Uparrow_{m\to i} U_m \quad = U_i\\
&\Uparrow_{m\to i} (I_m\ i) \quad = I_i\ (\mathsf{inj}_2\ i)\\
&\Uparrow_{m\to i} (!_m\ i) \quad = !_i\ i\\
&\Uparrow_{m\to i} (F \oplus_m G) \;=\; (\Uparrow_{m\to i} F) \oplus_i (\Uparrow_{m\to i} G)\\
&\Uparrow_{m\to i} (F \otimes_m G) \;=\; (\Uparrow_{m\to i} F) \otimes_i (\Uparrow_{m\to i} G)
\end{aligned}
$$

Occurrences under I_m can only be recursive types, never parameters, so we always inject them on the right for I_i.

We can also show the equivalence between the interpretations. We first need a way to lift the indexing function from `multirec` to `indexed`. We do this by applying the same function on the right—on the left we have zero parameters, so we match on the absurd pattern:

$$
\begin{aligned}
&\uparrow\_ : \{I : \mathsf{Set}\} \to \mathsf{Indexed}_m\ I \to \mathsf{Indexed}_i\ (\bot \uplus I)\\
&(\uparrow r)\ (\mathsf{inj}_1\ ())\\
&(\uparrow r)\ (\mathsf{inj}_2\ i) \;=\; r\ i
\end{aligned}
$$

The equivalence of the interpretations (without considering fixed points) follows:

$$
\begin{aligned}
&\uparrow_{m\to i} : \{I : \mathsf{Set}\}\ \{r : \mathsf{Indexed}_m\ I\}\ \{i : I\}\\
&\qquad (C : \mathsf{Code}_m\ I) \to [\![\ C\ ]\!]_m\ r\ i \equiv [\![\ \Uparrow_{m\to i} C\ ]\!]_i\ (\uparrow r)\ i\\
&\uparrow_{m\to i} U_m \quad = \mathsf{refl}\\
&\uparrow_{m\to i} (I_m\ i) \quad = \mathsf{refl}\\
&\uparrow_{m\to i} (!_m\ i) \quad = \mathsf{refl}\\
&\uparrow_{m\to i} (F \oplus_m G) \;=\; \mathsf{cong}_2\ \_\uplus\_\ (\uparrow_{m\to i} F)\ (\uparrow_{m\to i} G)\\
&\uparrow_{m\to i} (F \otimes_m G) \;=\; \mathsf{cong}_2\ \_\times\_\ (\uparrow_{m\to i} F)\ (\uparrow_{m\to i} G)
\end{aligned}
$$

We omit the proof with fixed points, since it relies on the same techniques used in the following section, where we show the proofs in more detail.

8.2.5 Indexed to InstantGenerics

We now show how to convert from a fixed-point view to the coinductive representation of `instant-generics` (subscript ig). Since all fixed-point views embed into `indexed`,

we need to define only the embedding of `indexed` into `instant-generics`. Since the two universes are less similar, the code transformation requires more care:

$$
\begin{aligned}
&\Uparrow_{i\to ig} : \{I\ O : \mathsf{Set}\} \to \mathsf{Code}_i\ I\ O \to (I \to \mathsf{Code}_{ig}) \to (O \to \mathsf{Code}_{ig}) \\
&\Uparrow_{i\to ig} U_i \quad\ \ r\ o = U_{ig} \\
&\Uparrow_{i\to ig} (I_i\ i) \quad\ \ r\ o = r\ i \\
&\Uparrow_{i\to ig} (!_i\ i) \quad\ \ r\ o = K_{ig}\ (o \equiv i) \\
&\Uparrow_{i\to ig} (F \oplus_i G)\ r\ o = (\Uparrow_{i\to ig} F\ r\ o) \oplus_{ig} (\Uparrow_{i\to ig} G\ r\ o) \\
&\Uparrow_{i\to ig} (F \otimes_i G)\ r\ o = (\Uparrow_{i\to ig} F\ r\ o) \otimes_{ig} (\Uparrow_{i\to ig} G\ r\ o) \\
&\Uparrow_{i\to ig} (F \circledcirc_i G)\ r\ o = R_{ig}\ (\sharp\ \Uparrow_{i\to ig} F\ (\Uparrow_{i\to ig} G\ r)\ o) \\
&\Uparrow_{i\to ig} (\mathsf{Fix}_i\ F)\ \ r\ o = R_{ig}\ (\sharp\ \Uparrow_{i\to ig} F\ [\, r\ ,\ \Uparrow_{i\to ig} (\mathsf{Fix}_i\ F)\ r\,]\ o) \\
&\Uparrow^{\mathsf{Set}}_{i\to ig} : \{I\ O : \mathsf{Set}\} \to \mathsf{Code}_i\ I\ O \to (I \to \mathsf{Set}) \to (O \to \mathsf{Code}_{ig}) \\
&\Uparrow^{\mathsf{Set}}_{i\to ig} C\ r\ o\ = \Uparrow_{i\to ig} C\ (K_{ig} \circ r)\ o
\end{aligned}
$$

We use two code conversion functions. $\Uparrow^{\mathsf{Set}}_{i\to ig}$ is the user-visible function, since it takes a parameter r that converts input indices into Sets. However, during the conversion we need to know how input indices map into Code_{ig}s; for this we use the $\Uparrow_{i\to ig}$ function.

Unit, sum, and product exist in both universes, so their conversion is trivial. Recursive invocations with I_i are handled by r, which tells how to map indices to Code_{ig}s. We lose the ability to abstract over recursive positions, which is in line with the behavior of `instant-generics`. Tagging is converted to a constant, trivially inhabited if we are in the expected output index o, and empty otherwise. Note that since an `indexed` code can define multiple types, but an `instant-generics` code can only represent one type, $\Uparrow_{i\to ig}$ effectively produces multiple `instant-generics` codes, one for each output index of the original `indexed` family.

A composition $F \circledcirc_i G$ is encoded through recursion; the resulting code is the conversion of F, whose parameters are `indexed` G functors. A fixed-point $\mathsf{Fix}_i\ F$ is naturally encoded through recursion, in a similar way to composition. The recursive positions of the fixed point are either: parameters on the left, converted with r as before; or recursive occurrences on the right, handled by recursively converting the codes with $\Uparrow_{i\to ig}$ and interpreting.

Having the code conversion in place, we can proceed to convert values:

$$
\begin{aligned}
&\mathsf{from}_{i\to ig} : \{I\ O : \mathsf{Set}\}\ \{r : I \to \mathsf{Code}_{ig}\} \\
&\qquad (C : \mathsf{Code}_i\ I\ O)\ (o : O) \to [\![\, C \,]\!]_i\ ([\![\, \_ \,]\!]_{ig} \circ r)\ o \to [\![\ \Uparrow_{i\to ig} C\ r\ o\ ]\!]_{ig} \\
&\mathsf{from}_{i\to ig}\ U_i \quad\ \ o\ \mathsf{tt} \quad\ \ = \mathsf{tt}_{ig} \\
&\mathsf{from}_{i\to ig}\ (I_i\ i) \quad\ \ o\ x \quad\ \ = x \\
&\mathsf{from}_{i\to ig}\ (!_i\ i) \quad\ \ o\ x \quad\ \ = k_{ig}\ x \\
&\mathsf{from}_{i\to ig}\ (F \oplus_i G)\ o\ (\mathsf{inj}_1\ x) = \mathsf{inj}_{1\,ig}\ (\mathsf{from}_{i\to ig}\ F\ o\ x) \\
&\mathsf{from}_{i\to ig}\ (F \oplus_i G)\ o\ (\mathsf{inj}_2\ x) = \mathsf{inj}_{2\,ig}\ (\mathsf{from}_{i\to ig}\ G\ o\ x) \\
&\mathsf{from}_{i\to ig}\ (F \otimes_i G)\ o\ (x\ ,\ y) \quad = (\mathsf{from}_{i\to ig}\ F\ o\ x)\ ,_{ig}\ (\mathsf{from}_{i\to ig}\ G\ o\ y) \\
&\mathsf{from}_{i\to ig}\ (F \circledcirc_i G)\ o\ x \quad\ \ = \mathsf{rec}_{ig}\ (\mathsf{from}_{i\to ig}\ F\ o\ (\mathsf{map}_i\ F\ (\mathsf{from}_{i\to ig}\ G)\ o\ x)) \\
&\mathsf{from}_{i\to ig}\ (\mathsf{Fix}_i\ F)\ \ o\ \langle\, x \,\rangle_i \quad = \mathsf{rec}_{ig}\ (\mathsf{from}_{i\to ig}\ F\ o\ (\mathsf{map}_i\ F\ [\, \mathsf{id}_{\rightrightarrows i}\ ,\ \mathsf{from}_{i\to ig}\ (\mathsf{Fix}_i\ F)\,]\ o\ x))
\end{aligned}
$$

```
from^Set_i→ig : {I O : Set} (C : Code_i I O) (r : I → Set) (o : O) →
                [[ C ]]_i r o → [[ ⇑^Set_i→ig C r o ]]_ig
from^Set_i→ig C r o = from_i→ig {r = K_ig ∘ r} C o ∘ map_i C (λ _ → k_ig) o
```

Again we use two functions to account for the fact that user datatypes know only how to map input indices into Sets, and not into Code_{ig}s. Function $\mathsf{from}^{\mathsf{Set}}_{i\to ig}$ invokes $\mathsf{from}_{i\to ig}$ after performing the necessary wrapping of indices with k_{ig}. The cases for composition and fixed point are the most interesting because we have to map the conversion function inside the argument positions; we do this using the map_i function. As usual, the inverse function $\mathsf{to}_{i\to ig}$ is entirely symmetrical, so we omit it.

Note that in the type of $\mathsf{from}_{i\to ig}$ we instantiate the function which interprets indices (the r argument) with an `instant-generics` interpretation. However, r has type $\mathsf{I} \to \mathsf{Set}$, whereas $[\![\_]\!]_{ig}$ has return type Set_1. If we were to raise `indexed` to Set_1, the interpretation function would then have type $\mathsf{I} \to \mathsf{Set}_1$, but then we could no longer use it in the I_i case. For now we rely on the Agda flag `--type-in-type`, and leave a formally correct solution for future work (see Section 8.4).

It remains to show that the conversion functions form an isomorphism. We start with the two most interesting cases: composition and fixed points. Following Section 6.2, we use lifted composition, equality, and identity to natural transformations in `indexed` (respectively $\_ \circ_{\rightrightarrows_i} \_$, $\_ \equiv_{\rightrightarrows_i} \_$, and $\mathsf{id}_{\rightrightarrows_i}$). We use equational reasoning for the proofs:

```
iso¹_i→ig : {I O : Set} (C : Code_i I O)
            (r : I → Code_ig) → (to_i→ig {r = r} C ∘_⇉i from_i→ig C) ≡_⇉i id_⇉i
iso¹_i→ig (F ⊙_i G) r o x =
  begin
    map_i F (to_i→ig G) o (to_i→ig F o (from_i→ig F o (map_i F (from_i→ig G) o x)))
  ≡⟨ cong (map_i F (to_i→ig G) o) (iso¹_i→ig F _ o _) ⟩
    map_i F (to_i→ig G) o (map_i F (from_i→ig G) o x)
  ≡⟨ sym (map°_i F (to_i→ig G) (from_i→ig G) o x) ⟩
    map_i F (to_i→ig G ∘_⇉i from_i→ig G) o x
  ≡⟨ map^∀_i F (iso¹_i→ig G r) o x ⟩
    map_i F id_⇉i o x
  ≡⟨ map^id_i F o x ⟩
    x ∎
```

The proof for composition is relatively simple, relying on applying the proof recursively, fusing the two maps, and reasoning by recursion on the resulting map, which results in an identity map. The proof for fixed points is slightly more involved:

```
iso¹_i→ig (Fix_i F) r o ⟨ x ⟩_i = cong ⟨_⟩_i $
  begin
    map_i F [ id_⇉i , to_i→ig (Fix_i F) ] o
```

```
      (to_{i→ig} F ∘ (from_{i→ig} F ∘ (map_i F [ id_{⇉i} , from_{i→ig} (Fix_i F) ] ∘ x)))
≡⟨ cong (map_i F [ id_{⇉i} , to_{i→ig} (Fix_i F) ] ∘) (iso1_{i→ig} F _ ∘ _) ⟩
  map_i F [ id_{⇉i} , to_{i→ig} (Fix_i F) ] ∘ (map_i F [ id_{⇉i} , from_{i→ig} (Fix_i F) ] ∘ x)
≡⟨ sym (map°_i F [ id_{⇉i} , to_{i→ig} (Fix_i F) ] [ id_{⇉i} , from_{i→ig} (Fix_i F) ] ∘ x) ⟩
  map_i F ([ id_{⇉i} , to_{i→ig} (Fix_i F) ] ∘_{⇉i} [ id_{⇉i} , from_{i→ig} (Fix_i F) ]) ∘ x
≡⟨ sym (map^∀_i F [,]∘_i ∘ x) ⟩
  map_i F [ id_{⇉i} ∘_{⇉i} id_{⇉i} , to_{i→ig} (Fix_i F) ∘_{⇉i} from_{i→ig} (Fix_i F) ] ∘ x
≡⟨ map^∀_i F ([,]cong_i (λ _ _ → refl) (iso1_{i→ig} (Fix_i F) r)) ∘ x ⟩
  map_i F [ id_{⇉i} , id_{⇉i} ] ∘ x
≡⟨ map^∀_i F [,]id_i ∘ x ⟩
  map_i F id_{⇉i} ∘ x
≡⟨ map^id_i F ∘ x ⟩
  x ∎
```

We start in the same way as with composition, but once we have fused the maps we have to deal with the fact that we are mapping distinct functions to the left (arguments) and right (recursive positions). We proceed with a lemma on disjunctive maps that states that a composition of disjunctions is the disjunction of the compositions ($[,]\circ_i$). Then we are left with a composition of identities on the left, which we solve with reflexivity, and a composition of $to_{i\to ig}$ and $from_{i\to ig}$ on the right, which we solve by induction. Finally, we show that a disjunction of identities is the identity (with the $[,]id_i$ lemma), and that the identity map is the identity. Note that we use the functor laws from Section 6.2, whose types are repeated here for convenience:

```
map^∀_i : {I O : Set} {r s : Indexed_i I} {f g : r ⇉_i s} (C : Code_i I O) →
          f ≡_{⇉i} g → map_i C f ≡_{⇉i} map_i C g
map°_i  : {I O : Set} {r s t : Indexed_i I} (C : Code_i I O) (f : s ⇉_i t) (g : r ⇉_i s) →
          map_i C (f ∘_{⇉i} g) ≡_{⇉i} (map_i C f ∘_{⇉i} map_i C g)
map^id_i : {I O : Set} {r : Indexed_i I} (C : Code_i I O) →
           map_i {r = r} C id_{⇉i} ≡_{⇉i} id_{⇉i}
```

Note also that $[,]\circ_i$, $[,]id_i$, and $[,]cong_i$ are similar to $||\circ_i$, $||id_i$, and $||cong_i$ from Section 6.2, respectively, but stated in terms of $[\_,\_]$ instead of $\_||_i\_$.

The proof for the remaining codes is unsurprising:

```
iso1_{i→ig} U_i          r ∘ _      = refl
iso1_{i→ig} (I_i i)      r ∘ _      = refl
iso1_{i→ig} (!_i i)      r ∘ _      = refl
iso1_{i→ig} (F ⊕_i G) r ∘ (inj_1 x) = cong inj_1 (iso1_{i→ig} F r ∘ x)
iso1_{i→ig} (F ⊕_i G) r ∘ (inj_2 x) = cong inj_2 (iso1_{i→ig} G r ∘ x)
iso1_{i→ig} (F ⊗_i G) r ∘ (x , y)   = cong_2 _,_ (iso1_{i→ig} F r ∘ x) (iso1_{i→ig} G r ∘ y)
```

8.3 Applying generic functions across approaches

The main operational advantage of our comparison is that it allows us to apply generic functions from approach `B` to representations of approach `A`, as long as `A` embeds into `B`. In this section we show some examples of that functionality.

In Section 6.1.3 we introduced an `indexed` code 'ListF' for lists (which we name 'ListF'$_\mathsf{i}$ in this section). Using that code we can define an example list in `indexed`:

```
aList_i : ⟦ 'ListF'_i ⟧_i (const ⊤) tt
aList_i = ⟨ inj₂ (tt , ⟨ inj₂ (tt , ⟨ inj₂ (tt , ⟨ inj₁ tt ⟩_i) ⟩_i) ⟩_i) ⟩_i
```

This encodes the exact same list as $\mathsf{aList}_{\mathsf{ig}}$ from Section 7.1. We can then apply the $\mathsf{size}_{\mathsf{ig}}$ function (defined in the same section) to $\mathsf{aList}_{\mathsf{i}}$, after converting using the functions shown in Section 8.2.5:

```
test_i→ig : size_ig (⇑^Set_i→ig 'ListF'_i _ _) (from^Set_i→ig 'ListF'_i _ _ aList_i) ≡ 4
test_i→ig = refl
```

Recall that $\mathsf{size}_{\mathsf{ig}}$ takes two arguments: first the code, which is the result of converting the list code 'ListF'$_\mathsf{i}$ to `instant-generics` with $\Uparrow^{\mathsf{Set}}_{\mathsf{i}\to\mathsf{ig}}$, and then the value, which we convert with $\mathsf{from}_{\mathsf{i}\to\mathsf{ig}}$. The parameters regarding `indexed` indices can be figured out by Agda, so we replace them by an underscore.

Recall now the Zig/Zag family of mutually recursive datatypes introduced in Section 5.1. We have defined many generic functions in the `indexed` universe, and argued in Chapter 6 that these could be applied to the other approaches shown so far. For instance, we can apply decidable equality to the zigZagEnd value:

```
test^aux_m→i : Maybe (_ ≡ _)
test^aux_m→i = deqt_i (Fix_i (⇑_m→i ZigZagC_m)) (λ _ _ → λ ()) _ x x
  where x = fromμ_m→i _ _ zigZagEnd_m

test_m→i : test^aux_m→i ≡ just refl
test_m→i = refl
```

Note also that the embedding relation is transitive. The following example brings the encoding of a natural number aNat from `regular` (Section 3.1) into `indexed`, and traverses it with a catamorphism, effectively doubling the number:

```
test^aux_r→i : ℕ
test^aux_r→i = cata_i _ f _ (fromμ_p→i _ (fromμ_r→p _ aNat_r))
  where f : ⊤ → ⊤ ⊎ ℕ → ℕ
        f _ (inj₁ _) = 0
        f _ (inj₂ n) = 2 + n

test_r→i : test^aux_r→i ≡ 4
test_r→i = refl
```

The ability to apply generic functions across approaches means that the functions only have to be defined in one universe, and can be applied to data from any embeddable

universe. This is particularly relevant for advanced generic behaviour, such as the zipper, which can be complicated to define in each universe. In effect, by defining an `indexed` zipper in Section 6.4, we have also defined a zipper for the `regular`, `polyp`, and `multirec` approaches, using the code conversions defined in this chapter.

8.4 Discussion

In this chapter we have compared different generic programming universes by showing the inclusion relation between them. This is useful to determine that one approach can encode at least as many datatypes as another approach, and also allows for lifting representations between compatible approaches. This also means that generic functions from approach `B` are all applicable in approach `A`, if `A` embeds into `B`, because we can bring generic values from approach `A` into `B` and apply the function there. Efficiency concerns remain to be investigated and addressed.

Our comparison does not allow us to make statements about the variety of generic functions that can be encoded in each approach. The generic map function, for instance, cannot be defined in `instant-generics`, while it is standard in `indexed`. One possible direction for future research is to devise a formal framework for evaluating what generic functions are possible in each universe, adding another dimension to our comparison of approaches.

Notably absent from our comparison are libraries with a generic view not based on a sum of products. In particular, the spine view [Hinze et al., 2006] is radically different from the approaches we model; yet, it is the basis for a number of popular generic programming libraries. It would be interesting to see how these approaches relate to those we have seen, but, at the moment, converting between entirely different universes remains a challenge.

An issue that remains with our modelling is to properly address termination. While our conversion functions can be used operationally to enable portability across different approaches, to serve as a formal proof they have to be terminating. Since Agda's algorithm for checking termination is highly syntax-driven, attempts to convince Agda of termination are likely to clutter the model, making it less easy to understand. We have thus decided to postpone such efforts for future work, perhaps relying on sized types for guaranteeing termination of our proofs [Abel, 2010].

A related issue that remains to be addressed is our use of `--type-in-type` in Section 8.2.5 for the code conversion function. It is not clear how to solve this issue even with the recently added support for universe polymorphism in Agda.

Nonetheless, we believe that this work is an important first step towards a formal categorisation of generic programming libraries. Future approaches can rely on our formalisation to describe precisely the new aspects they introduce, and how the new approach relates to existing ones. In this way we can hope for a future of *modular generic programming*, where a specific library might be constructed using components from different approaches, tailored to a particular need while still reusing code from existing libraries.

Part II

Practical aspects of generic programming

CHAPTER 9

Optimisation of generic programs

In the first part of this thesis we have introduced multiple libraries for generic programming in Haskell. In Chapter 8 we have compared them in terms of expressiveness, but we have not looked at how fast they perform. It is widely believed that generic programs run slower than type-specific handwritten code, given the conversions to and from representation types, and this factor often prevents adoption of generic programming altogether. On the other hand, generic functions can be specialised to particular datatypes, removing any overhead from the use of generic programming. In this chapter we analyse how to improve the performance of generic programs in GHC. We choose a representative library and look at the generated core code for a number of example generic functions. After understanding the necessary compiler optimisations for producing efficient generic code, we benchmark the runtime of our generic functions against handwritten variants, and conclude that all the overhead can indeed be removed.

9.1 Introduction

The performance of generic programs has been analysed before. Rodriguez Yakushev et al. [2008] present a detailed comparison of nine libraries for generic programming, with a brief performance analysis. This analysis indicates that the use of a generic approach could result in an increase of the running time by a factor of as much as 80. Van Noort et al. [2010] also report severe performance degradation when comparing a generic approach to a similar but type-specific variant. While this is typically not a problem for smaller examples, it can severely impair adoption of generic programming in larger contexts. This problem is particularly relevant because generic programming techniques are especially applicable to large applications where performance is crucial,

such as structure editors or compilers.

To understand the source of performance degradation when using a generic function from a particular generic programming library, we have to analyse the implementation of the library. The fundamental idea behind generic programming is to represent all datatypes by a small set of representation types. Equipped with conversion functions between user datatypes and their representation, we can define functions on the representation types, which are then applicable to all user types via the conversion functions. While these conversion functions are typically trivial and can be automatically generated, the overhead they impose is not automatically removed. In general, conversions to and from the generic representations are not eliminated by compilation, and are performed at run-time. These conversions are the source of inefficiency for generic programming libraries. In the earlier implementations of generic programming as code generators or preprocessors [Hinze et al., 2007], optimisations (such as automatic generation of type-specialised variants of generic functions) could be implemented externally. With the switch to library approaches, all optimisations have to be performed by the compiler, as the library approach no longer generates code itself.

GHC compiles a program by first converting the input into a core language and then transforming the core code into more optimised versions, in a series of sequential passes. While it performs a wide range of optimisations, with the default settings it seems to be unable to remove the overhead incurred by using generic representations. Therefore generic libraries perform slower than handwritten type-specific counterparts. Alimarine and Smetsers [2004, 2005] show that in many cases it is possible to remove all overhead by performing a specific form of symbolic evaluation in the Clean language. In fact, their approach is not restricted to optimising generics, and GHC performs symbolic evaluation as part of its optimisations. Our goal is to convince GHC to optimise generic functions so as to achieve the same performance as handwritten code, without requiring any additional manipulation of the compiler internals.

We have investigated this problem before [Magalhães et al., 2010b], and concluded that tweaking GHC optimisation flags can achieve significant speedups. The problem with using compiler flags is that these apply to the entire program being compiled, and while certain flags might have a good effect on generic functions, they might adversely affect performance (or code size) of other parts of the program. In this chapter we take a more fine-grained approach to the problem, looking at how to localise our performance annotations to the generic code only, by means of rewrite rules and function pragmas.[1] In this way we can improve the performance of generic functions with minimal impact on the rest of the program.

We continue this chapter by defining two representative generic functions which we focus our optimisation efforts on (Section 9.2). We then see how these functions can be optimised manually (Section 9.3), and transfer the necessary optimisation techniques to the compiler (Section 9.4). We confirm that our optimisations result in better run-time performance of generic programs in a benchmark in Section 9.5, and conclude in Section 9.6.

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html`

9.2 Example generic functions

For analysing the performance of generic programs we choose a simple but representative generic programming library from those presented in Part I, namely `instant-generics` (Chapter 7). We reuse the Haskell encoding of Section 7.2, and present two generic functions that will be the focus of our attention: equality and enumeration.

9.2.1 Generic equality

A notion of structural equality can easily be defined as a generic function. We start by giving the user-facing class and equality method:

```
class GEq α where
  geq :: α → α → Bool
```

This class is similar to the Prelude Eq class, but we have left out inequality for simplicity.

Adhoc instances for base types can reuse the Prelude implementation:

```
instance GEq Int where
  geq = (≡)
```

To be able to give generic instances for types such as lists, we first define a class for equality on the representation types:

```
class GEqRep α where
  geqRep :: α → α → Bool
```

We can now give instances for each of the representation types:

```
instance GEqRep U where
  geqRep _ _ = True
instance (GEqRep α, GEqRep β) ⇒ GEqRep (α :+: β) where
  geqRep (L x) (L y) = geqRep x y
  geqRep (R x) (R y) = geqRep x y
  geqRep _     _     = False
instance (GEqRep α, GEqRep β) ⇒ GEqRep (α :×: β) where
  geqRep (x1 :×: y1) (x2 :×: y2) = geqRep x1 x2 ∧ geqRep y1 y2
```

Units are trivially equal. For sums we continue the comparison recursively if both values are either on the left or on the right, and return False otherwise. Products are equal if both components are equal.

For variables and recursion we use the GEq class we have defined previously:

```
instance (GEq α) ⇒ GEqRep (Rec α) where
  geqRep (Rec x) (Rec y) = geq x y
```

```
instance (GEq α) ⇒ GEqRep (Var α) where
  geqRep (Var x) (Var y) = geq x y
```

The generic equality function, to be used when defining the geq method for generic instances, simply applies geqRep after converting the values to their generic representation:

```
geqDefault :: (Representable α, GEqRep (Rep α)) ⇒ α → α → Bool
geqDefault x y = geqRep (from x) (from y)
```

Now we can give a generic instance for lists:

```
instance (GEq α) ⇒ GEq [α] where
  geq = geqDefault
```

9.2.2 Generic enumeration

We now define a function that enumerates all possible values of a datatype. For infinite datatypes we have to make sure that every possible value will eventually be produced. For instance, if we are enumerating integers, we should not first enumerate all positive numbers, and then the negatives. Instead, we should interleave positive and negative numbers.

While equality is a generic consumer, taking generic values as input, enumeration is a generic producer, since it generates generic values. We enumerate values by listing them with the standard list type. There is only one unit to enumerate, and for variables and recursion we refer to the user-facing GEnum class as usual:

```
class GEnumRep α where
  genumRep :: [α]

instance GEnumRep U where
  genumRep = [U]
instance (GEnum α) ⇒ GEnumRep (Rec α) where
  genumRep = map Rec genum
instance (GEnum α) ⇒ GEnumRep (Var α) where
  genumRep = map Var genum
```

The more interesting cases are those for sums and products. For sums we enumerate both alternatives, but interleave them with a (|||) operator:

```
instance (GEnumRep α, GEnumRep β) ⇒ GEnumRep (α :+: β) where
  genumRep = map L genumRep ||| map R genumRep

infixr 5 |||
(|||) :: [α] → [α] → [α]
```

For products we generate all possible combinations of the two arguments, and diagonalise the result matrix, ensuring that all elements from each sublist will eventually be included, even if the lists are infinite:

```
instance (GEnumRep α, GEnumRep β) ⇒ GEnumRep (α :×: β) where
  genumRep = diag (map (\x → map (\y → x :×: y) genumRep) genumRep)

diag :: [[α]] → [α]
```

We omit the implementation details of (|||) and diag as they are not important; it only matters that we have some form of fair interleaving and diagonalisation operations. The presence of (|||) and diag throughout the generic function definition makes enumeration more complicated than equality, since equality does not make use of any auxiliary functions. We will see in Section 9.4.3 how this complicates the specialisation process. Note also that we do not use the more natural list comprehension syntax for defining the product instance, again to simplify the analysis of the optimisation process.

Finally we define the user-facing class, and a default implementation:

```
class GEnum α where
  genum :: [α]

genumDefault :: (Representable α, GEnumRep (Rep α)) ⇒ [α]
genumDefault = map to genumRep
```

9.3 Specialisation, by hand

We now focus on the problem of specialisation of generic functions. By specialisation we mean removing the use of generic conversion functions and representation types, replacing them by constructors of the original datatype. To convince ourselves that this task is possible we first develop a hand-written derivation of specialisation by equational reasoning. For simplicity we ignore implementation mechanisms such as the use of type classes and type families, and focus first on a very simple datatype encoding natural numbers:

```
data Nat = Ze | Su Nat
```

We give the representation of naturals using a type synonym:

```
type RepNat = U :+: Nat
```

We use a shallow representation (with Nat at the leaves, and not RepNat), remaining faithful with `instant-generics`. We also need a way to convert between RepNat and Nat:

```
toNat :: RepNat → Nat
toNat n = case n of
            (L U) → Ze
            (R n) → Su n

fromNat :: Nat → RepNat
fromNat n = case n of
              Ze     → L U
              (Su n) → R n
```

We now analyse the specialisation of generic equality and enumeration on this datatype.

9.3.1 Generic equality

We start with a handwritten, type-specific definition of equality for Nat:

```
eqNat :: Nat → Nat → Bool
eqNat m n = case (m , n) of
              (Ze    , Ze)    → True
              (Su m , Su n) → eqNat m n
              (_     , _)     → False
```

For equality on RepNat, we need equality on units and sums:

```
eqU :: U → U → Bool
eqU x y = case (x , y) of
            (U , U) → True

eqPlus :: (α → α → Bool) → (β → β → Bool) →
          α :+: β → α :+: β → Bool
eqPlus ea eb a b = case (a , b) of
                     (L x , L y) → ea x y
                     (R x , R y) → eb x y
                     (_   , _)   → False
```

Now we can define equality for RepNat, and generic equality for Nat through conversion to RepNat:

```
eqRepNat :: RepNat → RepNat → Bool
eqRepNat = eqPlus eqU eqNatFromRep

eqNatFromRep :: Nat → Nat → Bool
eqNatFromRep m n = eqRepNat (fromNat m) (fromNat n)
```

Our goal now is to show that eqNatFromRep is equivalent to eqNat. In the following derivation, we start with the definition of eqNatFromRep, and end with the definition of eqNat:

```
  eqRepNat (fromNat m) (fromNat n)

≡⟨ inline eqRepNat ⟩

  eqPlus equ eqNatFromRep (fromNat m) (fromNat n)

≡⟨ inline eqPlus ⟩

  case (fromNat m , fromNat n) of
    (L x , L y) → equ x y
    (R x , R y) → eqNatFromRep x y
    _           → False

≡⟨ inline fromNat ⟩

  case ( case m of {Ze → L U; Su x1 → R x1}
       , case n of {Ze → L U; Su x2 → R x2}) of
    (L x , L y) → equ x y
    (R x , R y) → eqNatFromRep x y
    _           → False

≡⟨ case-of-case transform ⟩

  case (m , n) of
    (Ze    , Ze)    → equ U U
    (Su x1 , Su x2) → eqNatFromRep x1 x2
    _               → False

≡⟨ inline equ and case-of-constant ⟩

  case (m , n) of
    (Ze    , Ze)    → True
    (Su x1 , Su x2) → eqNatFromRep x1 x2
    _               → False

≡⟨ inline eqNatFromRep , induction ⟩

  case (m , n) of
    (Ze    , Ze)    → True
    (Su x1 , Su x2) → eqNat x1 x2
    _               → False
```

This shows that the generic implementation is equivalent to the type-specific variant, and that it can be optimised to remove all conversions. We discuss the techniques

used in this derivation in more detail in Section 9.4.1, after showing the optimisation of generic enumeration.

9.3.2 Generic enumeration

A type-specific enumeration function for Nat follows:

```
enumNat :: [ Nat ]
enumNat  =  [Ze] ||| map Su enumNat
```

To get an enumeration for RepNat we first need to know how to enumerate units and sums:

```
enumU :: [ U ]
enumU  =  [U]
enumPlus :: [ α ] → [ β ] → [ α :+: β ]
enumPlus ea eb  =  map L ea ||| map R eb
```

Now we can define an enumeration for RepNat:

```
enumRepNat :: [ RepNat ]
enumRepNat  =  enumPlus enumU enumNatFromRep
```

With the conversion function toNat, we can use enumRepNat to get a generic enumeration function for Nat:

```
enumNatFromRep :: [ Nat ]
enumNatFromRep  =  map toNat enumRepNat
```

We now show that enumNatFromRep and enumNat are equivalent:

```
   map toNat enumRepNat

≡⟨ inline enumRepNat ⟩

   map toNat (enumPlus enumU enumNatFromRep)

≡⟨ inline enumPlus ⟩

   map toNat (map L enumU ||| map R enumNatFromRep)

≡⟨ inline enumU ⟩

   map toNat (map L [U] ||| map R enumNatFromRep)

≡⟨ inline map ⟩
```

```
    map toNat ([L U] ||| map R enumNatFromRep)
≡⟨ free theorem (|||) : ∀ f a b. map f (a ||| b) = map f a ||| map f b ⟩
    map toNat [L U] ||| map toNat (map R enumNatFromRep)
≡⟨ inline map ⟩
    [toNat (L U)] ||| map toNat (map R enumNatFromRep)
≡⟨ inline toNat and case-of-constant ⟩
    [Ze] ||| map toNat (map R enumNatFromRep)
≡⟨ functor composition law: ∀ f g l. map f (map g l) = map (f ∘ g) l ⟩
    [Ze] ||| map (toNat ∘ R) enumNatFromRep
≡⟨ inline toNat and case-of-constant ⟩
    [Ze] ||| map Su enumNatFromRep
```

Like equality, generic enumeration can also be specialised to a type-specific variant without any overhead.

9.4 Specialisation, by the compiler

After the manual specialisation of generic functions, let us now analyse how to convince the compiler to automatically perform the specialisation.

9.4.1 Optimisation techniques

Our calculations in Section 9.3 rely on a number of lemmas and techniques that the compiler will have to use. We review them here:

Inlining Inlining replaces a function call with its definition. It is a crucial optimisation technique because it can expose other optimisations. However, inlining causes code duplication, and care has to be taken to avoid non-termination through infinite inlining.

GHC uses a number of heuristics to decide when to inline a function or not, and loop breakers for preventing infinite inlining [Peyton Jones and Marlow, 2002]. The programmer can provide explicit inlining annotations with the INLINE and NOINLINE pragmas, of the form:

```
{−# INLINE [n] f #−}
```

In this pragma, f is the function to be inlined, and n is a phase number. GHC performs a number of optimisation phases through a program, numbered in decreasing order until zero. Setting n to 1, for instance, means "be keen to inline f in phase 1 and after". For a NOINLINE pragma, this means "do not inline f in phase 1 or after". The phase can be left out, in which case the pragma applies to all phases.[2]

Application of free theorems and functor laws Free theorems [Wadler, 1989] are theorems that arise from the type of a polymorphic function, regardless of the function's definition. Each polymorphic function is associated with a free theorem, and functions with the same type share the same theorem. The functor laws (mentioned in Section 6.2 for the `indexed` universe) arise from the categorical nature of functors. Every Functor instance in Haskell should obey the functor laws.

GHC does not compute and use the free theorem of each polymorphic function, also because it may not be clear which direction of the theorem is useful for optimisation purposes. However, we can add special optimisation rules to GHC via a RULES pragma [Peyton Jones et al., 2001]. For instance, the rewrite rule corresponding to the free theorem of (|||) follows:

```
{−# RULES "ft |||" forall f a b. map f (a ||| b) = map f a ||| map f b #−}
```

This pragma introduces a rule named "ft |||" telling GHC to replace appearances of the application map f (a ||| b) with map f a ||| map f b. GHC does not perform any confluence checking on rewrite rules, so the programmer should ensure confluence or GHC might loop during compilation.

Optimisation of case statements Case statements drive evaluation in GHC's core language, and give rise to many possible optimisations. Peyton Jones and Santos [1998] provide a detailed account of these; in our derivation in Section 9.3.2 we used a "case of constant" rule to optimise a statement of the form:

```
case (L ()) of
  L () → Ze
  R n → Su n
```

Since we know what we are case-analysing, we can replace this case statement by the much simpler expression Ze. Similarly, in Section 9.3.1 we used a case-of-case transform to eliminate an inner case statement. Consider an expression of the form:

```
case (case x of {e1 → e2}) of
  e2 → e3
```

[2] See the GHC User's Guide for more details: `http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html`.

Here, e_1, e_2, and e_3 are expressions starting with a constructor. We can simplify this to:

```
case x of
  e1 → e3
```

This rule naturally generalises to case statements with multiple branches.

9.4.2 Generic equality

We have seen that we have a good number of tools at our disposal for directing the optimisation process in GHC: inline pragmas, rewrite rules, phase distinction, and all the standard optimisations for the functional core language. We will now annotate our generic functions and evaluate the quality of the core code generated by GHC.

We start by defining a Representable instance for the Nat type:

```
instance Representable Nat where
  type Rep Nat = U :+: Rec Nat

  to (L U)       = Ze
  to (R (Rec n)) = Su n

  from Ze     = L U
  from (Su n) = R (Rec n)
```

We can now provide a generic definition of equality for Nat:

```
instance GEq Nat where
  geq = geqDefault
```

Compiling this code with the standard optimisation flag `-O`, and using `-ddump-simpl` to output the generated core code after all the simplification passes, we get the following core code:

```
$GEqNat_geq :: Nat → Nat → Bool
$GEqNat_geq = λ (x :: Nat) (y :: Nat) → case x of
                                          Ze   → case y of
                                                   Ze   → True
                                                   Su m → False
                                          Su m → case y of
                                                   Ze   → False
                                                   Su n → $GEqNat_geq m n
```

The core language is a small, explicitly typed language in the style of System F [Yorgey et al., 2012]. The function $\$GEqNat_{geq}$ is prefixed with a $ because it was generated by the compiler, representing the geq method of the GEq instance for Nat. We can see that the generic representation was completely removed, even without any INLINE pragmas.

The same happens for lists, as evidenced by the generated core code:

```
$GEq[]geq :: forall α. GEq α ⇒ [α] → [α] → Bool
$GEq[]geq = λ α (eqA :: GEq α) (l1 :: [α]) (l2 :: [α]) →
  case l1 of
    []       → case l2 of
                 []      → True
                 (h : t) → False
    (h1 : t1) → case l2 of
                 []       → False
                 (h2 : t2) → case eqA h1 h2 of
                               False → False
                               True  → $GEq[]geq α eqA t1 t2
```

Note that type abstraction and application is explicit in core. There is syntax to distinguish type and value application and abstraction from each other, but we suppress the distinction since it is clear from the colour. Note also that constraints (to the left of the ⇒ arrow) become just ordinary parameters, so $\$GEq[]_{geq}$ takes a function to compute equality on the list elements, eqA.[3]

Perhaps surprisingly, GHC performs all the required steps of Section 9.3.1 without requiring any annotations. In general, however, we found that it is sensible to provide INLINE pragmas for each instance of the representation datatypes when defining a generic function. In the case of geqRep, the methods are small, so GHC inlines them eagerly. For more complicated generic functions, the methods may become larger, and GHC will avoid inlining them. Supplying an INLINE pragma tells GHC to inline the methods anyway. The same applies to the from and to methods; for Nat and lists these methods are small, but in general they should be annotated with an INLINE pragma.

9.4.3 Generic enumeration

Generic consumers, such as equality, are, in our experience, more easily optimised by GHC. A generic producer such as enumeration, in particular, is challenging because it requires map fusion, and lifting auxiliary functions through maps using free theorems. As such, we encounter some difficulties while optimising enumeration. We start by looking at the natural numbers:

```
instance GEnum Nat where
  genum = map to genumRep
```

Note that instead of using genumDefault we directly inline its definition; this is to circumvent a limitation in the current implementation of defaults that prevents later rewrite rules from applying. GHC then generates the following code:

[3] The type of eqA is GEq α, but we use it as if it had type α → α → Bool. In the generated core there is also a coercion around the use of eqA to transform the class type into a function. This is the standard way class methods are desugared into core; we elide these details as they are not relevant to optimisation itself.

```
$x2 :: [ U :+: Rec Nat ]
$x2 = map $x4 $GEnumNat_genum
$x1 :: [ U :+: Rec Nat ]
$x1 = $x3 ||| $x2
$GEnumNat_genum :: [ Nat ]
$GEnumNat_genum = map to $x1
```

We omit the definitions of x_3 and x_4 for brevity. To make progress we need to tell GHC to move the map to expression in $\$GEnumNat_{genum}$ through the (|||) operator. We use a rewrite rule for this:

```
{-# RULES "ft |||" forall f a b. map f (a ||| b) = map f a ||| map f b #-}
```

With this rule in place, GHC generates the following code:

```
$x2 :: [ U :+: Rec Nat ]
$x2 = map $x4 $GEnumNat_genum
$x1 :: [ Nat ]
$x1 = map to $x2
$GEnumNat_genum :: [ Nat ]
$GEnumNat_genum = $x3 ||| $x1
```

We now see that the x_1 term is map applied to the result of a map. The way map is optimised in GHC (by conversion to build/foldr form) interferes with our `"ft |||"` rewrite rule, and map fusion is not happening. We can remedy this with an explicit map fusion rewrite rule:

```
{-# RULES "map/map1" forall f g l. map f (map g l) = map (f ∘ g) l #-}
```

This rule results in much improved generated code:

```
$x3 :: [ U :+: Rec Nat ]
$x3 = $x4 : []
$x2 :: [ Nat ]
$x2 = map to $x3
$x1 :: [ Nat ]
$x1 = map Su $GEnumNat_genum
$GEnumNat_genum :: [ Nat ]
$GEnumNat_genum = $x2 ||| $x1
```

The only thing we are missing now is to optimise x_3; note that its type is `[ U :+: Rec Nat ]`, and not `[ Nat ]`. For this we simply need to tell GHC to eagerly map a function over a list with a single element:

```
{-# RULES "map/map2" forall f x. map f (x : []) = (f x) : [] #-}
```

With this, GHC can finally generate the fully specialised enumeration function on Nat:

```
$x2 :: [ Nat ]
$x2 = Ze : []
$x1 :: [ Nat ]
$x1 = map Su $GEnumNat_genum
$GEnumNat_genum :: [ Nat ]
$GEnumNat_genum = $x2 ||| $x1
```

Optimisation for lists proves to be more difficult. Since lists use products, we need to introduce a rewrite rule for the free theorem of diag, allowing map to be pushed inside diag:

```
{-# RULES "ft/diag" forall f l. map f (diag l) = diag (map (map f) l) #-}
```

With this rule, and the extra optimisation flag `-fno-full-laziness` to maximise the chances for rewrite rules to apply, we get the following code:

```
$GEnum[]_genum :: forall α. GEnum α ⇒ [ [ α ] ]
$GEnum[]_genum = λ (gEnumA :: GEnum α) →
  ([] : []) ||| let $x1 :: [ Rec [ α ] ]
                    $x1 = map Rec ($GEnum[]_genum gEnumA)
                in diag (map (λ ($x3 :: α) →
                              map (λ ($x2 :: Rec [ α ]) → case $x2 of
                                                            Rec $x4 → $x3 : $x4) $x1)
                             gEnumA)
```

Most of the generic overhead is optimised away, but one problem remains: $x1 maps Rec over the recursive enumeration elements, but this Rec is immediately eliminated by a **case** statement. If $x1 was inlined, GHC could perform a map fusion, and then eliminate the use of Rec altogether. However, we have no way to specify that $x1 should be inlined; the compiler generated it, so only the compiler can decide when to inline it. Also, we had to use the compiler flag `-fno-full-laziness` to prevent some let-floating, but the flag applies to the entire program and might have unintended side-effects.

Reflecting on our developments in this section, we have seen that:

- Convincing GHC to optimise genum for a simple datatype such as Nat requires the expected free theorem of (|||). However, due to interaction between phases of application of rewrite rules, we are forced to introduce new rules for optimisation of map.
- Optimising genum for a more complicated datatype like lists requires the expected free theorem of diag. However, even after further tweaking of optimisation flags, we are unable to derive a fully optimised implementation. In any case, the partial optimisation achieved is certainly beneficial.

- More generally, we see that practical optimisation of generic functions is hard because of subtle interactions between the different optimisation mechanisms involved, such as inlining, rewrite rule application, **let** floating, **case** optimisation, etc.

These experiments have been performed with the latest GHC version available at the time of writing, 7.4.1. We have also observed that the behavior of the optimiser changes between compiler versions. In particular, some techniques which resulted in better code in some versions (e.g. the use of SPECIALISE pragmas) result in worse code in other versions. This makes it practically impossible to devise a set of general guidelines for generic function optimisation; unfortunately, users requiring the best performance will have to inspect the generated core code and experiment with the possible optimisations.

9.5 Benchmarking

Having seen the inner workings of the GHC optimiser, we now verify experimentally that our optimisation techniques result in a runtime improvement of code using generic functions.

9.5.1 Benchmark suite design

Benchmarking is, in general, a complex task, and a lazy language imposes even more challenges on the design of a benchmark. We designed a benchmark suite that ensures easy repeatability of tests, calculating the average running time and the standard deviation for statistical analysis. It is portable across different operating systems and can easily be run with different compiler versions. It supports passing additional flags to the compiler and reports the flags used in the final results. Each test is compiled as an independent program, which consists of getting the current CPU time, running the test, getting the updated CPU time and outputting the time difference. The Unix `time` command performs a similar task, but unfortunately no equivalent is immediately available in Microsoft Windows. By including the timing code with the test (using the portable function System.CPUTime.getCPUtime) we work around this problem.

To ensure reliability of the benchmark we use profiling, which gives us information about which computations last longer. For each of the tests, we ensure that at least 50% of the time is spent on the function we want to benchmark. Profiling also gives information about the time it takes to run a program, but we do not use those figures since they are affected by the profiling overhead. An approach similar to the `nofib` benchmark suite [Partain, 1993] is not well-suited to our case, as our main point is not to compare an implementation across different compilers, but instead to compare different implementations on a single compiler.

A top-level Haskell script takes care of compiling all the tests with the same flags, invoking them a given number of times, parsing and accumulating results as each test finishes, and calculating and displaying the average running time at the end, along with some system information.

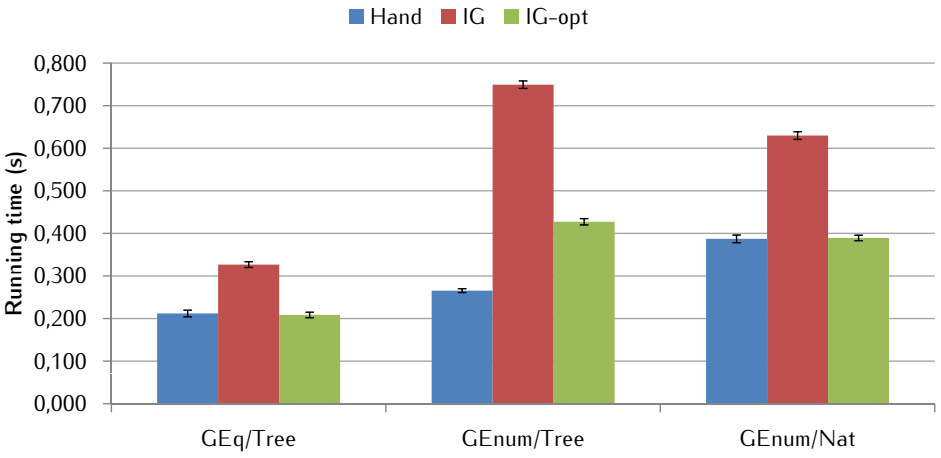


Figure 9.1: Running time comparison between handwritten, generic, and optimised generic code for three independent tests.

We have a detailed benchmark suite over different datatypes, generic functions, and generic programming libraries.[4] Here we show only a small subset of the results gathered from our benchmark, to confirm the insights gained from Section 9.4.

9.5.2 Results

We have tested generic equality and enumeration on a simple datatype of binary trees:

```
data Tree α = Bin α (Tree α) (Tree α) | Leaf
```

For enumeration we have also used the Nat datatype. The results can be seen in Figure 9.1. We compare the running time for each of three variants: handwritten, `instant-generics` without any optimisations, and `instant-generics` with the optimisations described previously, all compiled with `-O1`. For equality, we can see that using our optimisations removes all the overhead from generic programming, as we expected. Enumeration on Nat is also optimised to perfection, but for more complex datatypes, such as Tree, we run into the problems observed in Section 9.4.3; while the optimisations are beneficial, they are not enough to bring the performance to the same level as handwritten code.

[4] `https://subversion.cs.uu.nl/repos/staff.jpm.public/benchmark/trunk/`

9.6 Conclusion

In this chapter we have looked at the problem of optimising generic functions. With their representation types and associated conversions, generic programs tend to be slower than their type-specific handwritten counterparts, and this can limit adoption of generic programming in situations where performance is important. We have picked one specific library, `instant-generics`, that is representative of the type of generic programming we explore in this thesis, and investigated the code generation for generic programs, and the necessary optimisation techniques to fully remove any overhead from the library. We concluded that the overhead can be fully removed most of the time, using only already available optimisations that apply to functional programs in general. However, due to the difficulty of managing the interaction between several different optimisations, in some cases we are not able to fully remove the overhead. We are confident, however, that this is only a matter of further tweaking of GHC's optimisation strategies.

Some work remains to be done in terms of improving the user experience. We have mentioned that the to and from functions should be inlined; this should be automatically established by the mechanism for deriving Representable instances. Additionally, inserting INLINE pragmas for each case in the generic function is a tedious process, which should also be automated. Finally, it would be interesting to see if the definition of rewrite rules based on free theorems of auxiliary functions used could be automated; it is easy to generate free theorems, but it is not always clear how to use these theorems for optimisation purposes.

While `instant-generics` is a representative library for the approaches covered in this thesis, other approaches, such as Scrap Your Boilerplate [SYB, Lämmel and Peyton Jones, 2003, 2004], use different implementation mechanisms and certainly require different optimisation strategies. It remains to be seen how to optimise other approaches, and to establish general guidelines for optimisation of generic programs.

In any case, it is now clear that generic programs do not have to be slow, and their optimisation up to handwritten code performance is not only possible but also achievable using only standard optimisation techniques. This opens the door for a future where generic programs are not only general, elegant, and concise, but also as efficient as type-specific code.

CHAPTER 10

Generic programming for indexed datatypes

An indexed datatype is a type that uses a parameter as a type-level tag; a typical example is the type of vectors, which are indexed over a type-level natural number encoding their length. Since the introduction of GADTs, indexed datatypes have become commonplace in Haskell. Values of indexed datatypes are often more involved than values of plain datatypes, and programmers would benefit from having generic programs on indexed datatypes. However, no generic programming library in Haskell adequately supports them, leaving programmers with the tedious task of writing repetitive code.

In this chapter we show how to extend the `instant-generics` approach to deal with indexed datatypes. Our approach can also be used in similar libraries, and is fully backwards-compatible, meaning that existing code will continue working without modification. We show not only how to encode indexed datatypes generically, but also how to instantiate generic functions on indexed datatypes. Furthermore, all generic representations and instances are generated automatically, making life easier for users.

10.1 Introduction

Runtime errors are undesirable and annoying. Fortunately, the strong type system of Haskell eliminates many common programmer mistakes that lead to runtime errors, like unguarded casts. However, even in standard Haskell, runtime errors still occur often. A typical example is the error of calling `head` on an empty list.

Indexed datatypes, popular since the introduction of GADTs, allow us to avoid calling `head` on an empty list and many other runtime errors by encoding further information at the type level. For instance, one can define a type of lists with a known length, and then define `head` in such a way that it only accepts lists of length greater than zero.

This prevents the usual mistake by guaranteeing statically that head is never called on an empty list.

Unfortunately, generic programming and indexed datatypes do not mix well. The added indices and their associated type-level computations needs to be encoded in a generic fashion, and while this is standard in dependently-typed approaches to generic programming (for instance `indexed` of Chapter 6), we know of no generic programming approach in Haskell that deals with indexed datatypes. In fact, even the standard deriving mechanism, which automatically generates instances for certain type classes, fails to work for GADTs, in general.

We argue that it is time to allow these two concepts to mix. Driven by an application that makes heavy use of both generic programming and indexed datatypes [Magalhães and De Haas, 2011], we have developed an extension to `instant-generics` to support indexed datatypes. Our extension is both conservative, as it preserves all the functionality of `instant-generics` without requiring modifications of client code, and general, as it applies equally well to other generic programming libraries. Furthermore, we show that instantiating functions to indexed datatypes is not trivial, even in the non-generic case. In the context of datatype-generic programming, however, it is essential to be able to easily instantiate functions; otherwise, we lose the simplicity and reduced code duplication we seek. Therefore we show how to automatically instantiate generic functions to indexed datatypes, in a way that works for most types of generic functions.

We continue this chapter by first describing indexed datatypes in detail in Section 10.2. Section 10.3 describes how indexed datatypes can be represented generically, and Section 10.4 deals with instantiating generic functions to these datatypes. In Section 10.5 we sketch the algorithm for automatic representation of indexed datatypes and instantiation of generic functions. We then discuss how to deal with datatypes indexed over custom kinds in Section 10.6, review related work in Section 10.7, propose future work in Section 10.8, and conclude in Section 10.9.

10.2 Indexed datatypes

While the Haskell libraries described in the first part of this thesis already allow a wide range of datatypes to be handled in a generic fashion, they cannot deal with indexed datatypes.[1] We call a datatype *indexed* if it has a type parameter that is not used as data (also called a *phantom* type parameter), and at least one of the datatype's constructors introduces type-level constraints on this type. The type of vectors, or size-constrained lists, is an example of such a datatype:

```
data Vec α η where
  Nil  ::                      Vec α Ze
  Cons :: α → Vec α η → Vec α (Su η)
```

[1] The `indexed` approach of Chapter 6 can deal with such datatypes, but it remains unclear how to port this library to Haskell.

The first parameter of Vec, α, is the type of the elements of the vector. In the GADT syntax above with type signatures for each constructor, we see that α appears as an argument to the Cons constructor; α is a regular type parameter. On the other hand, the second parameter of Vec, η, does not appear as a direct argument to any constructor: it is only used to constrain the possible ways of building Vecs. We always instantiate η with the following empty (uninhabited) datatypes:

```
data Ze
data Su η

type T0 = Ze
type T1 = Su T0
type T2 = Su T1
```

A vector with two Chars, for instance, is represented as follows:

```
exampleVec :: Vec Char T2
exampleVec = Cons 'p' (Cons 'q' Nil)
```

Note that its type, Vec Char T_2, adequately encodes the length of the vector; giving any other type to exampleVec, such as Vec Char T_0 or Vec Char Char, would result in a type error.

Indexed types are easy to define as a GADT, and allow us to give more specific types to our functions. For instance, the type of vectors above allows us to avoid the usual `empty list` error when taking the first element of an empty list, since we can define a head function that does not accept empty vectors:

```
headVec :: Vec α (Su η) → α
headVec (Cons x _) = x
```

GHC correctly recognises that it is not necessary to specify a case for headVec Nil, since that is guaranteed never to happen by the type-checker.

Indexed datatypes are also useful when specifying well-typed embedded languages:

```
data Term α where
  Lit    :: Int                                  → Term Int
  IsZero :: Term Int                             → Term Bool
  Pair   :: Term α → Term β                      → Term (α , β)
  If     :: Term Bool → Term α → Term α → Term α
```

The constructors of Term specify the types of the arguments they require and the type of term they build. We will use the datatypes Vec and Term as representative examples of indexed datatypes in the rest of this chapter.

10.2.1 Type-level equalities and existential quantification

Indexed datatypes such as Vec and Term can be defined using only existential quantification and type-level equalities [Baars and Swierstra, 2004]. For example, GHC rewrites Vec to the following equivalent datatype:

```
data Vec α η  =          η ∼ Ze     ⇒ Nil
              | forall μ. η ∼ Su μ  ⇒ Cons α (Vec α μ)
```

The constructor Nil introduces the constraint that the type variable η equals Ze; $\alpha \sim \beta$ is GHC's notation for type-level equality between types α and β. The Cons constructor requires η to be a Su of something; this "something" is encoded by introducing an existentially-quantified variable μ, which stands for the length of the sublist, and restricting η to be the successor of μ (in other words, one plus the length of the sublist).

This encoding of Vec is entirely equivalent to the one shown previously. While it may seem more complicated, it makes explicit what happens "behind the scenes" when using the Nil and Cons constructors: Nil can only be used if η can be unified with Ze, and Cons introduces a new type variable μ, constrained to be the predecessor of η. In the next section we will look at how to encode indexed datatypes generically; for this we need to know what kind of primitive operations we need to support. Looking at this definition of Vec, it is clear that we will need not only a way of encoding type equality constraints, but also a way to introduce new type variables.

10.2.2 Functions on indexed datatypes

The extra type safety gained by using indexed datatypes comes at a price: defining functions operating on these types can be harder. Consumer functions are not affected; we can easily define an evaluator for Term, for instance:

```
eval :: Term α → α
eval (Lit i)      = i
eval (IsZero t)   = eval t ≡ 0
eval (Pair a b)   = (eval a , eval b)
eval (If p a b)   = if eval p then eval a else eval b
```

In fact, even GHC can automatically derive consumer functions on indexed datatypes for us; the following Show instances work as expected:

```
deriving instance Show α ⇒ Show (Vec α η)
deriving instance Show (Term α)
```

Things get more complicated when we look at producer functions. Let us try to define a function to enumerate values. For lists this is simple:

```
enumList :: [α] → [[α]]
enumList ea = [] : [x : xs | x ← ea, xs ← enumList ea]
```

Given an enumeration of all possible element values, we generate all possible lists, starting with the empty list.[2] However, a similar version for Vec is rejected by the compiler:

[2] We are not diagonalising the elements from ea and enumList ea, but that is an orthogonal issue.

```
enumVec :: [ α ] → [ Vec α η ]
enumVec ea = Nil : [Cons x xs | x ← ea, xs ← enumVec ea]
```

GHC complains of being unable to match Ze with Su η, and rightfully so: we try to add Nil, of type Vec α Ze, to a list containing Conses, of type Vec α (Su η). To make this work we can use type classes:

```
instance GEnum (Vec α Ze) where
  genum = [Nil]
instance (GEnum α, GEnum (Vec α η)) ⇒ GEnum (Vec α (Su η)) where
  genum = [Cons a t | a ← genum, t ← genum]
```

In this way we can provide different types (and implementations) to the enumeration of empty and non-empty vectors.

Note that GHC (version 7.4.1) is not prepared to derive producer code for indexed datatypes. Trying to derive an instance Read (Vec α η) results in the generation of type-incorrect code. The difficulties associated with deriving instances for indexed datatypes have been known for a while (see, for instance, the GHC bug reports #3012 and #4528). We show in Section 10.4.2 a way of identifying the necessary instances to be defined, which could also be used to improve instance deriving for GADTs.

10.3 Handling indexing generically

As we have seen in the previous section, to handle indexed datatypes generically we need support for type equalities and quantification in the generic representation. We deal with the former in Section 10.3.1, and the latter in Section 10.3.2.

10.3.1 Type equalities

A general type equality $\alpha \sim \beta$ can be encoded in a simple GADT:

```
data α :=: β where
  Refl :: α :=: α
```

We could add the :=: type to the representation types of `instant-generics`, and add type equalities as extra arguments to constructors. However, since the equalities are always introduced at the constructor level, and `instant-generics` has a representation type to encode constructors,[3] we prefer to define a more general representation type for constructors which also introduces a type equality:

```
data CEq γ ϕ ψ α where
  CEq :: α → CEq γ ϕ ϕ α
```

[3] We have elided it from the description in Chapter 7, but it is present in the distribution of the library. This way of encoding meta-information is explained in detail in Section 11.2.2.

The new CEq type takes two extra parameters which are forced to unify by the CEq constructor. The old behavior of C can be recovered by instantiating the ϕ and ψ parameters to trivially equal types:

```
type C γ α = CEq γ () () α
```

Note that we can encode multiple equalities as a product of equalities. For example, a constructor which introduces the equality constraints $\alpha \sim$ Int and $\beta \sim$ Char would be encoded with a representation of type CEq γ (α :×: β) (Int :×: Char) δ (for suitable γ and δ).

Encoding types with equality constraints

At this stage we are ready to encode types with equality constraints that do not rely on existential quantification; the :=: type shown before is a good example:

```
instance Representable (α :=: β) where
  type Rep (α :=: β) = CEq Eq_Refl α β U
  from Refl      = CEq U
  to   (CEq U)   = Refl
```

The type equality introduced by the Refl constructor maps directly to the equality introduced by CEq, and vice-versa. As Refl has no arguments, we encode it with the unit representation type U. The auxiliary datatype Eq_{Refl}, which we omit, is used to encode constructor information about Refl (see Section 11.2.2 for more information on encoding meta-information).

Generic functions over equality constraints

We need to provide instances for the new CEq representation type for each generic function. The instances for the equality and enumeration functions of Section 9.2 are:

```
instance (GEqRep α) ⇒ GEqRep (CEq γ φ ψ α) where
  geqRep (CEq a) (CEq b) = geqRep a b

instance (GEnumRep α) ⇒ GEnumRep (CEq γ φ φ α) where
  genumRep = map CEq genumRep
instance                GEnumRep (CEq γ φ ψ α) where
  genumRep = []
```

Generic consumers, such as equality, are generally not affected by the equality constraints. Generic producers are somewhat trickier, because we are now trying to build a generic representation, and thus must take care not to build impossible cases. For generic enumeration, we proceed normally in case the types unify, and return the empty enumeration in case the types are different. Note that these two instances overlap, but remain decidable.

10.3.2 Existentially-quantified indices

Recall the shape of the Cons constructor of the Vec datatype:

```
forall μ. η ~ Su μ  ⇒  Cons α (Vec α μ)
```

We need to be able to introduce new type variables in the type representation. A first idea would be something like:

```
type Rep (Vec α η)  =  forall μ. CEq Vec_Cons η (Su μ) ...
```

This however is not accepted by GHC, as the right-hand side of a type family instance cannot contain quantifiers. This restriction is well justified, as allowing this would lead to higher-order unification problems [Neubauer and Thiemann, 2002].

Another attempt would be to encode representations as data families instead of type families, so that we can use regular existential quantification:

```
data instance Rep (Vec α η)  =  forall μ. RepVec (CEq Vec_Cons η (Su μ) ...)
```

However, we do not want to use data families to encode the generic representation, as these introduce a new constructor per datatype, thereby effectively precluding a generic treatment of all types.

Faking existentials

Since the conventional approaches do not work, we turn to some more unconventional approaches. All we have is an index type variable η, and we need to generate existentially-quantified variables that are constrained by η. We know that we can use type families to create new types from existing types, so let us try that. We introduce a type family

```
type family X η
```

and we will use X η where the original type uses μ.[4] We can now write a generic representation for Vec:

```
instance Representable (Vec α η) where
  type Rep (Vec α η)  =  CEq Vec_Nil   η Ze          U
                      :+: CEq Vec_Cons η (Su (X η)) (Var α :×: Rec (Vec α (X η)))
  from Nil          =  L (CEq U)
  from (Cons h t)   =  R (CEq (Var h :×: Rec t))
  to (L (CEq U))                  =  Nil
  to (R (CEq (Var h :×: Rec t)))  =  Cons h t
```

This is a good start, but we are not done yet, as GHC refuses to accept the code above with the following error:

[4] This is closely related to Skolemization.

```
Could not deduce (m ~ X (Su m))
from the context (n ~ Su m)
bound by a pattern with constructor
  ConsVec :: forall a n. a -> Vec a n -> Vec a (Su n),
in an equation for 'from'
```

What does this mean? GHC is trying to unify μ with X (Su μ), when it only knows that $\eta \sim$ Su μ. The equality $\eta \sim$ Su μ comes from the pattern-match on Cons, but why is it trying to unify μ with X (Su μ)? Well, on the right-hand side we use CEq with type CEq $\mathsf{Vec}_{\mathsf{Cons}}$ η (Su (X η)) ..., so GHC tries to prove the equality $\eta \sim$ Su (X η). In trying to do so, it replaces η by Su μ, which leaves Su $\mu \sim$ Su (X (Su μ)), which is implied by $\mu \sim$ X (Su μ), but GHC cannot find a proof of the latter equality.

This is unsurprising, since indeed there is no such proof. Fortunately we can supply it by giving an appropriate type instance:

```
type instance X (Su μ) = μ
```

We call instances such as the one above "mobility rules", as they allow the index to "move" through indexing type constructors (such as Su) and X. Adding the type instance above makes the Representable instance for Vec compile correctly. Note also how X behaves much like an extraction function, getting the parameter of Su.

Representation for Term. The Term datatype (shown in Section 10.2) can be represented generically using the same technique. First let us write Term with explicit quantification and type equalities:

```
data Term α = α ~ Int                ⇒ Lit    Int
            | α ~ Bool               ⇒ IsZero (Term Int)
            | forall β γ. α ~ (β , γ) ⇒ Pair   (Term β) (Term γ)
            | If     (Term Bool) (Term α) (Term α)
```

We see that the Lit and IsZero constructors introduce type equalities, and the Pair constructor abstracts from two variables. This means we need two type families:

```
type family X₁ α
type family X₂ α
```

Since this strategy could require introducing potentially many type families, we use a single type family instead, parameterised over two other arguments:

```
type family X γ ι α
```

We instantiate the γ parameter to the constructor representation type, ι to a type-level natural indicating the index of the introduced variable, and α to the datatype index itself.

The representation for Term becomes:

```
type Rep_Term α = CEq Term_Lit    α Int
                      (Rec Int)
                :+: CEq Term_IsZero α Bool
                      (Rec (Term Int))
                :+: CEq Term_Pair   α (X Term_Pair T_0 α , X Term_Pair T_1 α)
                      (   Rec (Term (X Term_Pair T_0 α))
                      :×: Rec (Term (X Term_Pair T_1 α)))
                :+: C   Term_If
                      (Rec (Term Bool) :×: Rec (Term α) :×: Rec (Term α))
```

We show only the representation type $\mathsf{Rep}_{\mathsf{Term}}$, as the from and to functions are trivial. The mobility rules are induced by the equality constraint of the Pair constructor:

```
type instance X Term_Pair T_0 (β , γ) = β
type instance X Term_Pair T_1 (β , γ) = γ
```

Again, the rules resemble selection functions, extracting the first and second components of the pair.

Summarising, quantified variables are represented as type families, and type equalities are encoded directly in the new CEq representation type. Type equalities on quantified variables need mobility rules, represented by type instances. We have seen this based on two example datatypes; in Section 10.5 we describe more formally how to encode indexed datatypes in the general case.

10.4 Instantiating generic functions

Now that we know how to represent indexed datatypes, we proceed to instantiate generic functions on these types. We split the discussion into generic consumers and producers, as they require a different approach.

10.4.1 Generic consumers

Instantiating generic equality to the Vec and Term types is unsurprising:

```
instance (GEq α) ⇒ GEq (Vec α η) where
  geq = geqDefault
instance           GEq (Term α)  where
  geq = geqDefault
```

Using the instance for generic equality on CEq of Section 10.3.1, these instances compile and work fine. The instantiation of generic consumers on indexed datatypes is therefore no more complex than on standard datatypes.

10.4.2 Generic producers

Instantiating generic producers is more challenging, as we have seen in Section 10.2.2. For Vec, a first attempt could be:

```
instance (GEnum α) ⇒ GEnum (Vec α η) where
  genum = genumDefault
```

However, this will always return the empty list: GHC's instance resolution mechanism will look for a GEnumRep instance for the heads CEq γ Ze η β and CEq γ (Su μ) η β (for some γ and β which are not relevant to this example). In both cases, only the second GEnumRep instance of Section 10.3.1 applies, as neither Ze nor Su μ can be unified with η.

Therefore, and as before, we need to give two instances, one for Vec α Ze, and another for Vec α (Su η), given an instance for Vec α η:

```
instance (GEnum α) ⇒ GEnum (Vec α Ze) where
  genum = genumDefault
instance (GEnum α, GEnum (Vec α η)) ⇒ GEnum (Vec α (Su η)) where
  genum = genumDefault
```

We can check that this works as expected by enumerating all the vectors of Booleans of length one: genum :: [Vec Bool (Su Ze)] evaluates to [Cons True Nil, Cons False Nil], the two possible combinations.

Instantiating Term. Instantiating GEnum for the Term datatype follows a similar strategy. We must identify the types that Term is indexed on. These are Int, Bool, and (α , β), in the Lit, IsZero, and Pair constructors, respectively. The If constructor does not impose any constraints on the index, and as such can be ignored for this purpose. Having identified the possible types for the index, we give an instance for each of these cases:

```
instance GEnum (Term Int)  where
  genum = genumDefault
instance GEnum (Term Bool) where
  genum = genumDefault
instance (GEnum (Term α), GEnum (Term β)) ⇒ GEnum (Term (α , β)) where
  genum = genumDefault
```

We can now enumerate arbitrary Terms. However, having to write the three instances above manually is still a repetitive and error-prone task; while the method is trivial (simply calling genumDefault), the instance head and context still have to be given, but these are determined entirely by the shape of the datatype. We have written Template Haskell code to automatically generate these instances for the user.

In this section and the previous we have seen how to encode and instantiate generic functions for indexed datatypes. In the next section we look at how we automate this process, by analysing representation and instantiation in the general case.

10.5 General representation and instantiation

In general, an indexed datatype has the following shape:

$$\begin{aligned} \textbf{data}\ \mathsf{D}\ \overline{\alpha}\ = \ & \forall \overline{\beta_1}.\overline{\gamma_1}\ \Rightarrow\ \mathsf{C}_1\ \overline{\phi_1} \\ & \vdots \\ \mid\ & \forall \overline{\beta_n}.\overline{\gamma_n}\ \Rightarrow\ \mathsf{C}_n\ \overline{\phi_n} \end{aligned}$$

We consider a datatype D with arguments $\overline{\alpha}$ (which may or may not be indices), and n constructors $\mathsf{C}_1\ \ldots\ \mathsf{C}_n$, with each C_i constructor potentially introducing existentially-quantified variables $\overline{\beta_i}$, type equalities $\overline{\gamma_i}$, and a list of arguments $\overline{\phi_i}$. We use an overline to denote sequences of elements.

We need to impose some further restrictions to the types we are able to handle:

1. Quantified variables are not allowed to appear as standalone arguments to the constructor: $\forall_{i,\beta \in \overline{\beta_i}}.\beta \notin \overline{\phi_i}$.

2. Quantified variables have to appear in the equality constraints: $\forall_{i,\beta \in \overline{\beta_i}}.\exists\psi.\psi\ \beta \in \overline{\gamma_i}$. We require this to provide the mobility rules.

For such a datatype, we need to generate two types of code:

1. The generic representation

2. The instances for generic instantiation

We deal with (1) in Section 10.5.1 and (2) in Section 10.5.2.

10.5.1 Generic representation

Most of the code for generating the representation is not specific to indexed datatypes; see Section 11.4 for a definition of a similar representation. The code generation for constructors needs to be adapted from the standard definition, since now CEq takes two extra type arguments. The value generation (functions from and to) is not affected, only the representation type.

Type equalities. For each constructor C_i, an equality constraint $\tau_1 \sim \tau_2 \in \overline{\gamma_i}$ becomes the second and third arguments to CEq, for instance $\mathsf{CEq}\ \ldots\ \tau_1\ \tau_2\ \ldots$. Multiple constraints like $\tau_1 \sim \tau_2, \tau_3 \sim \tau_4$ become a product, as in $\mathsf{CEq}\ \ldots\ (\tau_1 :\times: \tau_3)\ (\tau_2 :\times: \tau_4)\ \ldots$. An existentially-quantified variable β_i appearing on the right-hand side of a constraint of the form $\tau \sim \ldots$ or on the arguments to C_i is replaced by $\mathsf{X}\ \gamma\ \iota\ \tau$, with γ the constructor representation type of C_i, and ι a type-level natural version of i.

Mobility rules. For the generated type families we provide the necessary mobility rules (Section 10.3.2). Given a constraint $\forall\overline{\beta_n}.\overline{\gamma_n}$, each equality $\beta \sim \psi\ \tau$, where $\beta \in \overline{\beta}$ and $\psi\ \tau$ is some type expression containing τ, we generate a **type instance** $X\ \gamma\ \iota\ (\psi\ \tau) = \tau$, where γ is the constructor representation type of the constructor where the constraint appears, and ι is a type-level natural encoding the index of the constraint. As an example, for the Cons constructor of Section 10.2.1, β is η, τ is μ, ψ is Su, γ is $\mathsf{Vec}_{\mathsf{Cons}}$, and ι is T_0 (since there is only one constraint).

10.5.2 Generic instantiation

To give a more thorough account of the algorithm for generating instances we sketch its implementation in Haskell. We assume the following representation of datatypes:

```
data Datatype = Datatype [TyVar] [Con]
data Con      = Con Constraints [Type]

data Type     -- abstract
data TyVar    -- abstract

tyVarType :: TyVar → Type
```

A datatype has a list of type variables as arguments, and a list of constructors. Constructors consist of constraints and a list of arguments. For our purposes, the particular representation of types and type variables is not important, but we need a way to convert type variables into types (tyVarType).

Constraints are a list of (existentially-quantified) type variables and a list of type equalities:

```
data Constraints = [TyVar] ⩾ [TyEq]
data TyEq        = Type :∼: Type
```

We will need equality on type equalities, so we assume some standard equality on types and type equalities.

We start by separating the datatype arguments into "normal" arguments and indices:

```
findIndices :: Datatype → [TyVar :+: TyVar]
findIndices (Datatype vs cs) = [if v `inArgs` cs then L v else R v | v ← vs]

inArgs :: TyVar → [Con] → Bool
inArgs = ...
```

We leave inArgs abstract, but its definition is straightforward: it checks if the argument TyVar appears in any of the constructors as an argument. In this way, findIndices tags normal arguments with L and potential indices with R. These are potential indices because they could also just be phantom types, which are not only not used as argument but also have no equality constraints. In any case, it is safe to treat them as indices. The isIndex function used before is defined in terms of findIndices:

```
isIndex :: TyVar → Datatype → Bool
isIndex t d = R t ∈ findIndices d
```

Having identified the indices, we want to identify all the return types of the constructors, as these correspond to the heads of the instances we need to generate. This is the task of function findRTs:

```
findRTs :: [TyVar] → [Con] → [Constraints]
findRTs is []                    = []
findRTs is ((Con cts args) : cs) = let rs = findRTs is cs
                                   in if anyIn is cts then cts : rs else rs

anyIn :: [TyVar] → Constraints → Bool
anyIn vs (_ ⩾ teqs) = or [v `in_TyEq` teqs | v ← vs]

in_TyEq :: TyVar → [TyEq] → Bool
in_TyEq = ...
```

We check the constraints in each constructor for the presence of a type equality of the form i :∼: t, for some index type variable i and some type t. We rely on the fact that GADTs are converted to type equalities of this shape; otherwise we should look for the symmetric equality t :∼: i too.

Having collected the important constraints from the constructors, we want to merge the constraints of the constructors with the same return type. Given the presence of quantified variables, this is not a simple equality test; we consider two constraints to be equal modulo all possible instantiations of the quantified variables:

```
instance Eq Constraints where
  (vs ⩾ cs) ≡ (ws ⩾ ds) = length vs ≡ length ws ∧ cs ≡ subst ws vs ds

subst :: [TyVar] → [TyVar] → [TyEq] → [TyEq]
subst vs ws teqs = ...
```

Two constraints are equal if they abstract over the same number of variables and their type equalities are the same, when the quantified variables of one of the constraints are replaced by the quantified variables of the other constraint. This replacement is performed by subst; we do not show its code since it is trivial (given a suitable definition of Type).

Merging constraints relies on constraint equality. Each constraint is compared to every element in an already merged list of constraints, and merged if it is equal:

```
merge :: Constraints → [Constraints] → [Constraints]
merge c1 [] = [c1]
merge c1@(vs ⩾ cs) (c2@(ws ⩾ ds) : css)
  | c1 ≡ c2 = c1 : css
  | otherwise = c2 : merge c1 css
```

```
mergeConstraints :: [Constraints] → [Constraints]
mergeConstraints = foldr merge []
```

We can now combine the functions above to collect all the merged constraints:

```
rightOnly :: [α :+: β] → [β]
rightOnly l = [x | R x ← l]

allConstraints :: Datatype → [Constraints]
allConstraints d@(Datatype _ cons) = let is = rightOnly (findIndices d)
                                     in mergeConstraints (findRTs is cons)
```

We know these constraints are of shape i :∼: t, where i is an index and t is some type. We need to generate instance heads of the form **instance** G (D $\overline{\alpha}$), where $\overline{\alpha} \in$ buildInsts D. The function buildInsts computes a list of type variable instantiations starting with the list of datatype arguments, and instantiating them as dictated by the collected constraints:

```
buildInsts :: Datatype → [[Type]]
buildInsts d@(Datatype ts _) = map (instVar ts) cs
  where cs = concat (map (λ(_ :≫: t) → t) (allConstraints d))

instVar :: [TyVar] → TyEq → [Type]
instVar []       _ = []
instVar (v : vs) tEq@(i :∼: t)
  | tyVarType v ≡ i = t : map tyVarType vs
  | otherwise       = tyVarType v : instVar vs tEq
```

This completes our algorithm for generic instantiation to indexed datatypes. As mentioned before, the same analysis could be used to find out what Read instances are necessary for a given indexed datatype. Note however that this algorithm only finds the instance head for an instance; the method definition is trivial when it is a generic function.

10.6 Indices of custom kinds

So far we have only considered datatypes with indices of kind ⋆. However, a recent GHC extension allows programmers to define their own kinds, paving the way for GADTs that are indexed over indices of kind other than ⋆ [Yorgey et al., 2012]. Custom indices allow for more type (or kind) safety. The type of vectors introduced in Section 10.2, for instance, can be given a more precise definition using data kinds:

```
data Nat = Ze | Su Nat
data Vec α :: Nat → ⋆ where
  Nil  ::                  Vec α Ze
  Cons :: α → Vec α η → Vec α (Su η)
```

Note the subtle difference: previously we defined two empty datatypes Ze and Su η, of kind $\star$, and used those as indices of Vec. With the `-XDataKinds` language extension, we can instead define a regular Nat datatype for natural numbers, with constructors Ze and Su. By means of *promotion*, the Nat declaration also introduces a new kind Nat, inhabited by two types, Ze and Su.[5] We can then give a more detailed kind to Vec, restricting its second argument (an index) to be of the newly-defined kind Nat. The return type of the Nil constructor, for instance, is Vec α Ze, where Ze has kind Nat, and not $\star$ as in Section 10.2.

An immediate advantage of using datakinds for indices is that it reduces the potential for errors. The type Vec Int Bool, for instance, now gives rise to a kind error, since Bool has kind $\star$, and not Nat. The Term datatype of Section 10.2 can also be encoded with datakinds:

```
data Term_i = Int_i | Bool_i | Pair_i Term_i Term_i
data Term :: Term_i → ⋆ where
  Lit    :: Int                          → Term Int_i
  IsZero :: Term Int_i                   → Term Bool_i
  Pair   :: Term α → Term β              → Term (Pair_i α β)
  If     :: Term Bool_i → Term α → Term α → Term α
```

We define a new datatype Term_i standing for the type of indices of Term. We are not really interested in the values of Term_i, only in its promotion; we use the promoted kind Term_i in the kind of Term, and the promoted constructors in the types of the constructors of Term.

The mechanism to deal with indexed datatypes that we have explained so far focuses on indices of kind $\star$. Fortunately, with one small change, it can equally well deal with other kinds. Recall the kind of the X representation type from Section 10.3.2:

```
type family X (γ :: ⋆) (ι :: ⋆) (α :: ⋆) :: ⋆
```

It states that the kind of datatype indices, α, is $\star$. This is too restrictive if we want to use indices of kind Nat or Term_i. We could define a new type family for each datatype to represent, with the appropriate kind for the α parameter, but this is repetitive and obscures the fact that X should be independent of the datatype being represented. Fortunately, there is also a GHC extension for *kind polymorphism*, which is exactly what we need for defining X. Its kind becomes:

```
type family X (γ :: ⋆) (ι :: Nat) (α :: k) :: k
```

We abstract from the kind of indices with a kind variable k. Since X will replace an index, its kind has to be the same as that of the index, so the return kind of X is k. Note

[5] Since types and constructors can have the same name in Haskell, there might be ambiguity when resolving a potentially promoted name. In source code, promoted constructors can be prefixed with a single quote as a means to resolve the ambiguity. In our presentation, we use colours to distinguish between syntactic categories.

also that we can improve the kind of ι, the natural number used to distinguish between multiple indexed variables.

Changing the kind of X is all that it takes to support datakinds as indices with our approach; the remainder of the description, including the instantiation of generic functions, remains unchanged.

10.7 Related work

Indexed datatypes can be seen as a subset of all GADTs, or as existentially-quantified datatypes using type-level equalities. Johann and Ghani [2008] developed a categorical semantics of GADTs, including an initial algebra semantics. While this allows for a better understanding of GADTs from a generic perspective, it does not translate directly to an intuitive and easy-to-use generic library.

Gibbons [2008] describes how to view abstract datatypes as existentially-quantified, and uses final coalgebra semantics to reason about such types. Rodriguez Yakushev and Jeuring [2010] describe an extension to the spine view [Hinze et al., 2006] supporting existential datatypes. Both approaches focus on existentially-quantified *data*, whereas we do not consider this case at all, instead focusing on (potentially existentially-quantified) *indices*. See Section 10.8 for a further discussion on this issue.

Within dependently-typed programming, indexing is an ordinary language feature which can be handled generically more easily due to the presence of type-level lambdas and explicit type application, as we have seen in Section 6.1.8.

10.8 Future work

While we can express indexed datatypes both as GADTs and as existentially-quantified datatypes with type-level equalities, the reverse is not true in general. Consider the type of dynamic values:

```
data Dynamic = forall α. Typeable α ⇒ Dyn α
```

The constructor Dyn stores a value on which we can use the operations of type class Typeable, which is all we know about this value. In particular, its type is not visible "outside", since Dynamic has no type variables. Another example is the following variation of Term:

```
data Term α where
  Const :: α                    → Term α
  Pair  :: Term α → Term β      → Term (α , β)
  Fst   :: Term (α , β)         → Term α
  Snd   :: Term (α , β)         → Term β
```

Here, the type argument α is not only an index but it is also used as data, since values of its type appear in the Const constructor. Our approach cannot currently deal with such

datatypes. We plan to investigate if we can build upon the work of Rodriguez Yakushev and Jeuring [2010] to also support existentials when used as data.

10.9 Conclusion

In this chapter we have seen how to increase the expressiveness of a generic programming library by adding support for indexed datatypes. We used the `instant-generics` library for demonstrative purposes, but we believe the technique readily generalises to all other generic programming libraries using type-level generic representation and type classes. We have seen how indexing can be reduced to type-level equalities and existential quantification. The former are easily encoded in the generic representation, and the latter can be handled by encoding the restrictions on the quantified variables as relations to the datatype index. All together, our work brings the convenience and practicality of datatype-generic programming to the world of indexed datatypes, widely used in many applications but so far mostly ignored by generic programming approaches.

CHAPTER 11

Integrating generic programming in Haskell

Haskell's **deriving** mechanism supports the automatic generation of instances for a number of functions. The Haskell 98 Report [Peyton Jones, 2003] only specifies how to generate instances for the Eq, Ord, Enum, Bounded, Show, and Read classes. The description of how to generate instances is largely informal, and it imposes restrictions on the shape of datatypes, depending on the particular class to derive. As a consequence, the portability of instances across different compilers is not guaranteed.

In this chapter we describe a new approach to Haskell's **deriving** mechanism in which instance derivation is realised through default methods. This allows users to define classes whose instances require only a single line of code specifying the class and the datatype, similarly to the way **deriving** currently works. We use this to leverage generic programming in Haskell; generic functions, including the methods from the above six Haskell 98 derivable classes, can then be specified in Haskell itself, making them lightweight and portable.

11.1 Introduction

Generic programming has come a long way: from its roots in category theory, passing through dedicated languages, language extensions and pre-processors until the flurry of library-based approaches of today (Section 8.1). In this evolution, expressivity has not always increased: many generic programming libraries of today still cannot compete with Generic Haskell, for instance. The same applies to performance, as libraries tend to do little regarding code optimisation, whereas meta-programming techniques such as Template Haskell [Sheard and Peyton Jones, 2002] can generate near-optimal code. Instead, generic programming techniques seem to evolve in the direction of better

availability and usability: it should be easy to define generic functions and it should be trivial to use them. Certainly some of the success of the SYB approach is due to its availability: it comes with GHC, the main Haskell compiler, which can even derive the necessary type class instances to make everything work with no effort.

To improve the usability of generics in Haskell, we believe a tighter integration with the compiler is necessary. In fact, the Haskell 98 standard already contains some generic programming, in the form of derived instances [Peyton Jones, 2003, Chapter 10]. Unfortunately, the report does not formally specify how to derive instances, and it restricts the classes that can be derived to six only (Eq, Ord, Enum, Bounded, Show, and Read). GHC has since long extended these with Data and Typeable (the basis of SYB), and more recently with Functor, Foldable, and Traversable. Due to the lack of a unifying formalism, these extensions are not easily mimicked in other compilers, which need to reimplement the instance code generation mechanism.

To address these issues we:

- Design a new generic programming library for Haskell. Our library borrows heavily from other approaches to generic programming in Haskell, but pays particular attention to practical usage concerns. In particular, it can represent the most common forms of datatypes, and encode the most commonly used generic functions, while remaining practical to use.

- Show how this library can be used to replace the **deriving** mechanism in Haskell, and provide some examples, such as the Functor class.

- Provide a detailed description of how the representation for a datatype is generated. In particular, we can represent almost all Haskell 98 datatypes.

- Implemented our approach in both the Utrecht Haskell Compiler [UHC, Dijkstra et al., 2009], and in GHC. The implementation in UHC reflects an earlier version of our proposal, and makes use of very few language extensions. The implementation in GHC uses type families in a way similar to `instant-generics` (Section 7.2). We also provide a package that shows in detail the code that needs to be added to the compiler, the code that should be generated by the compiler, and the code that is portable between compilers.[1]

This chapter is an updated version of previous work [Magalhães et al., 2010a]. We describe the design as it is implemented in GHC.[2]

We continue this chapter by introducing the generic programming library designed for our extension (Section 11.2). We proceed to show how to define generic functions (Section 11.3), and then describe the necessary modifications to the compiler for supporting our approach (Section 11.4). In Section 11.5 we explain the differences between the current description and our earlier design, which is implemented in UHC. Finally, we

[1] `http://hackage.haskell.org/package/generic-deriving`

[2] See also the description on the GHC user's guide: `http://www.haskell.org/ghc/docs/7.4.1/html/users_guide/generic-programming.html`.

discuss alternative designs (Section 11.6), review related work (Section 11.7), propose future work (Section 11.8), and conclude in Section 11.9.

11.2 The `generic-deriving` library

We begin with an example of what we want generic programming usage to look like in Haskell. We use the generic function encode as a running example throughout this chapter. This function transforms a value into a sequence of bits:

```
data Bit = O | I
class Encode α where
  encode :: α → [Bit]
```

A user should be able to define a datatype and ask for a generic implementation of encode, similarly to what happens for show:

```
data Exp = Const Int | Plus Exp Exp
  deriving (Show, Encode)
```

However, in our proposal we do not use **deriving** for generating instances. Instead, we propose to use an empty instance head:

```
instance Encode Exp
```

Using a special form of default method declaration in the Encode class (which we discuss in detail in Section 11.3.3), this empty instance definition gives rise to an instance of a generic implementation of encode for the type Exp. The function is then ready to be used:

```
test :: [Bit]
test = encode (Plus (Const 1) (Const 2))
```

This should be all that is necessary to use encode. The user should need no further knowledge of generics, and encode can be used in the same way as show, for instance.

Behind the scenes, the compiler uses a generated representation of Exp to instantiate a generic definition of encode. For this we require a generic representation, which we now describe. We discuss alternative designs and motivate our choice in more detail in Section 11.6. The new library, which we call `generic-deriving`, is a mix between `polyp` (Chapter 4) and `instant-generics` (Chapter 7), in the sense that it supports one datatype parameter, but it does not have a fixed-point view on data.

11.2.1 Run-time type representation

The choice of a run-time type representation affects not only the compiler writer, who has to implement the automatic generation of representations, but also the expressiveness of

the whole approach. A simple representation is easier to derive, but might not allow for defining certain generic functions. More complex representations are more expressive, but require more work for the automatic derivation of instances.

We present a set of representation types that tries to balance these factors. We use the common sum-of-products representation without explicit fixpoints but with explicit abstraction over a single parameter. Therefore, representable types are functors, and we can compose types. Additionally, we provide useful types for encoding meta-information (such as constructor names) and tagging arguments to constructors. We show examples of how these representation types are used in Section 11.2.4.

The basic ingredients of our representation are units, sums, and products:

```
data U1          ρ = U1
data (:+:) φ ψ ρ = L1 {unL1 :: φ ρ} | R1 {unR1 :: ψ ρ}
data (:×:) φ ψ ρ = φ ρ :×: ψ ρ
```

Like in the implementation of `polyp` (Section 4.2), we use a type parameter ρ to encode one datatype parameter. We also have lifted void (V_1) to represent nullary sums, but for simplicity we omit it from this discussion and from the generic functions in the next section.

We use an explicit combinator to mark the occurrence of the parameter:

```
newtype Par1 ρ = Par1 {unPar1 :: ρ}
```

As our representation is functorial, we can encode composition. We require the first argument of composition to be a representable type constructor (see Section 11.4 for more details). The second argument can only be the parameter, a recursive occurrence of a functorial datatype, or again a composition. We use Rec_1 to represent recursion, and :○: for composition:

```
newtype Rec1 φ    ρ = Rec1 {unRec1 :: φ ρ}
newtype (:○:) φ ψ ρ = Comp1 (φ (ψ ρ))
```

This is similar to the way `polyp` treats composition, but now without explicit fixed points.

Finally, we have two types for tagging and representing meta-information:

```
newtype K1 ι γ    ρ = K1 {unK1 :: γ}
newtype M1 ι γ φ ρ = M1 {unM1 :: φ ρ}
```

We use K_1 for tagging and M_1 for storing meta-information. The ι parameter is used for grouping different types of meta-information together, as can be seen in the following type synonyms:

```
data D                    type D1     = M1 D
data C                    type C1     = M1 C
data S                    type S1     = M1 S

data R                    type Rec0   = K1 R
data P                    type Par0   = K1 P
```

We use Rec_0 to tag occurrences of (possibly recursive) types of kind $\star$ and Par_0 to mark additional parameters of kind $\star$. For meta-information, we use D_1 for datatype information, C_1 for constructor information, and S_1 for record selector information. We group five combinators into two because in many generic functions the behavior is independent of the meta-information or tags. In this way, fewer trivial cases have to be given. For instance, generic map (Section 11.3.4) has a single instance for the M_1 representation type, while generic show (Section 11.3.6) has separate instances for D_1, C_1, and S_1. We present the meta-information associated with M_1 in detail in the next section.

Note that we abstract over a single parameter ρ of kind $\star$. This means we will be able to express generic functions such as

```
fmap :: (α → β) → ϕ α → ϕ β
```

but not

```
bimap :: (α → γ) → (β → δ) → ϕ α β → ϕ γ δ
```

For bimap we would need a type representation that can distinguish between the parameters. All representation types would need to carry one additional type argument. However, in practice few generic functions require abstraction over more than a single type parameter, so we support abstraction over one parameter only.

11.2.2 Meta-information

Some generic functions need information about datatypes, constructors, and records. This information is stored in the type representation:

```
class Datatype γ where
  datatypeName :: σ γ (ϕ :: ⋆ → ⋆) ρ → String
  moduleName   :: σ γ (ϕ :: ⋆ → ⋆) ρ → String

class Constructor γ where
  conName :: σ γ (ϕ :: ⋆ → ⋆) ρ → String
  conFixity :: σ γ (ϕ :: ⋆ → ⋆) ρ → Fixity
  conFixity = const Prefix
  conIsRecord :: σ γ (ϕ :: ⋆ → ⋆) ρ → Bool
  conIsRecord = const False
```

```
class Selector γ where
  selName :: σ γ (φ :: ⋆ → ⋆) ρ → String
```

Names are unqualified. We provide the datatype name together with the name of the module in which it is defined. This is the only meta-information we store for a datatype, although it could be easily extended to add the kind, for example. We only store the name of a selector. For a constructor, we also store its fixity and record if it has fields. This last information is not strictly necessary, as it can be inferred by looking for non-empty selNames, but it simplifies some generic function definitions. The datatypes Fixity and Associativity are unsurprising:

```
data Fixity = Prefix | Infix Associativity Int
data Associativity = LeftAssociative | RightAssociative | NotAssociative
```

We provide default definitions for conFixity and conIsRecord to simplify instantiation for prefix constructors that do not use record notation.[3]

The types of the meta-information functions are defined so that they match the type of the representation types. We provide more details in Section 11.4.5, and the examples later in Section 11.2.4 and Section 11.3.6 also clarify how these classes are used.

Note that we could encode the meta information as an extra argument to M_1:

```
data M1 ι φ ρ = M1 Meta (φ ρ)
data Meta = Meta String Fixity ...
```

However, with this encoding it is harder to write generic producers, since to produce an M_1 we have to produce a Meta for which we have no information. With the above representation we avoid this problem by using type-classes to fill in the right information for us. Section 11.3.5 shows an example of how this works.

11.2.3 A generic view on data

We obtain a generic view on data by defining an embedding-projection pair between a datatype and its type representation. We use the following classes for this purpose:

```
class Generic α where
  type Rep α :: ⋆ → ⋆
  from :: α → Rep α χ
  to   :: Rep α χ → α

class Generic1 φ where
  type Rep1 φ :: ⋆ → ⋆
```

[3] Note that all constructor arguments will be wrapped in an S_1, independently of using record notation or not. We omit this in the example representations of this section for space reasons, but it becomes clear in Section 11.4.

```
from1 :: φ ρ → Rep1 φ ρ
to1   :: Rep1 φ ρ → φ ρ
```

We use a type family to encode the representation of a standard type. In $\mathsf{Generic}_1$ we encode types of kind $\star \to \star$, so we have the parameter ρ. In Generic there is no parameter, so we invent a variable χ which is never used.

All types need to have an instance of Generic. Types of kind $\star \to \star$ also need an instance of $\mathsf{Generic}_1$. This separation is necessary because some generic functions (like fmap or traverse) require explicit abstraction from a single type parameter, whereas others (like show or enum) do not. Given the different kinds involved, it is unavoidable to have two type classes for this representation. Note, however, that we have a single set of representation types (apart from the duplication for tagging recursion and parameters).

11.2.4 Example representations

We now show how to represent some standard datatypes. Note that all the code in this section is automatically generated by the compiler, as described in Section 11.4.

Representing Exp. The meta-information for datatype Exp (defined in the beginning of this section) looks as follows:

```
data $Exp
data $Const_Exp
data $Plus_Exp

instance Datatype $Exp where
  moduleName _ = "ModuleName"
  datatypeName _ = "Exp"

instance Constructor $Const_Exp where conName _ = "Const"
instance Constructor $Plus_Exp  where conName _ = "Plus"
```

In moduleName, `"ModuleName"` is the name of the module where Exp lives. The particular datatypes we use for representing the meta-information at the type-level are not needed for defining generic functions, so they are not visible to the user. In this description we prefix them with a \$.

The type representation ties the meta-information to the sum-of-products representation of Exp:

```
type Rep0^Exp = D1 $Exp (    C1 $Const_Exp (Rec0 Int)
                        :+: C1 $Plus_Exp  (Rec0 Exp :×: Rec0 Exp))
```

Note that the representation is shallow: at the recursive occurrences we use Exp, and not $\mathsf{Rep}_0^{\mathsf{Exp}}$.

The embedding-projection pair implements the isomorphism between Exp and $\mathsf{Rep}_0^{\mathsf{Exp}}$:

```
instance Generic Exp where
  type Rep Exp  =  Rep_0^Exp
  from (Const n)   =  M_1 (L_1 (M_1 (K_1 n)))
  from (Plus e e')  =  M_1 (R_1 (M_1 (K_1 e :×: K_1 e')))
  to (M_1 (L_1 (M_1 (K_1 n))))            =  Const n
  to (M_1 (R_1 (M_1 (K_1 e :×: K_1 e')))) =  Plus e e'
```

Here it is clear that from and to are inverses: the pattern of from is the same as the expression in to, and vice-versa.

Representing lists. The representation for a type of kind $\star \to \star$ requires an instance for both $\mathsf{Generic}_1$ and Generic. For the standard Haskell list type [] we generate the following code:

```
type Rep_0^List ρ  =  D_1 $List (    C_1 $Nil_List    U_1
                                 :+: C_1 $Cons_List (Par_0 ρ :×: Rec_0 [ ρ ]))

instance Generic [ ρ ] where
  type Rep [ ρ ]  =  Rep_0^List ρ
  from []      =  M_1 (L_1 (M_1 U_1))
  from (h : t)  =  M_1 (R_1 (M_1 (K_1 h :×: K_1 t)))
  to (M_1 (L_1 (M_1 U_1)))            =  []
  to (M_1 (R_1 (M_1 (K_1 h :×: K_1 t)))) =  h : t
```

We omit the definitions for the meta-information, which are similar to the previous example. We use Par_0 to tag the parameter ρ, as we view lists as a kind $\star$ datatype for Generic. This is different in the $\mathsf{Generic}_1$ instance:

```
type Rep_1^List  =  D_1 $List (    C_1 $Nil_List    U_1
                               :+: C_1 $Cons_List (Par_1 :×: Rec_1 []))

instance Generic_1 [] where
  type Rep_1 []  =  Rep_1^List
  from_1 []      =  M_1 (L_1 (M_1 U_1))
  from_1 (h : t)  =  M_1 (R_1 (M_1 (Par_1 h :×: Rec_1 t)))
  to_1 (M_1 (L_1 (M_1 U_1)))            =  []
  to_1 (M_1 (R_1 (M_1 (Par_1 h :×: Rec_1 t)))) =  h : t
```

We treat parameters and recursion differently in $\mathsf{Rep}_0^{\mathsf{List}}$ and $\mathsf{Rep}_1^{\mathsf{List}}$. In $\mathsf{Rep}_0^{\mathsf{List}}$ we use Par_0 and Rec_0 for mere tagging; in $\mathsf{Rep}_1^{\mathsf{List}}$ we use Par_1 and Rec_1 instead, which store the parameter and the recursive occurrence of a type constructor, respectively. We will see later when defining generic functions (Section 11.3) how these are used.

Note that while there is still some code duplication, because there are two instances for generic representations, we do not have to duplicate the entire set of representation

types, since we can represent a datatype without parameters with our lifted representation.

Representing type composition. We now present a larger example, involving more complex datatypes, to show the expressiveness of our approach. Datatype Expr represents abstract syntax trees of a small language:

```
infixr 6 :*:
data Expr ρ = Const Int
            | Expr ρ :*: Expr ρ
            | EVar {unVar :: Var ρ}
            | Let [Decl ρ] (Expr ρ)
data Decl ρ = Decl (Var ρ) (Expr ρ)
data Var  ρ = Var ρ | VarL (Var [ρ])
```

Note that Expr makes use of an infix constructor (:*:), has a selector (unVar), and uses lists in Let. Datatype Var is nested, since in the VarL constructor Var is called with [ρ]. These oddities are present only for illustrating how our approach represents them. We show only the essentials of the encoding of this set of mutually recursive datatypes, starting with the meta-information:

```
data $:*:Expr
data $EVarExpr
data $UnVar

instance Constructor $:*:Expr where
  conName     _ = ":*:"
  conFixity   _ = Infix RightAssociative 6
instance Constructor $EVarExpr where
  conName     _ = "EVar"
  conIsRecord _ = True
instance Selector $UnVar where
  selName     _ = "unVar"
```

We have to store the fixity of the :*: constructor, and also the fact that EVar has a record. We store its name in the instance for Selector, and tie the meta-information to the representation:

```
type Rep1Expr = D1 $Expr
              (    (    C1 $ConstExpr (Rec0 Int)
                   :+: C1 $:*:Expr    (Rec1 Expr :×: Rec1 Expr))
              :+:  (    C1 $EVarExpr  (S1 $UnVar (Rec1 Var))
                   :+: C1 $LetExpr    (([] :○: Rec1 Decl) :×: Rec1 Expr)))
```

In $\mathsf{Rep}_1^{\mathsf{Expr}}$ we see the use of S_1. Also interesting is the representation of the Let constructor: the list datatype is applied not to the parameter ρ but to Decl ρ, so we use composition to denote this. Note also that we are using a balanced encoding for the sums (and also for the products). This improves the performance of the type-checker, and makes generic encoding more space-efficient.

We omit the representation for Decl. For Var we use composition again:

```
type Rep_1^Var = D_1 $Var (    C_1 $Var_Var  Par_1
                           :+: C_1 $VarL_Var (Var :o: Rec_1 []))
```

In the representation of the VarL constructor, Var is applied to $[\,\rho\,]$. We represent this as a composition with Rec_1 [].

When we use composition, the embedding-projection pairs become slightly more complicated:

```
instance Generic_1 Expr where
  type Rep_1 Expr = Rep_1^Expr
  from_1 (Const i)    = M_1 (L_1 (L_1 (M_1 (K_1 i))))
  from_1 (e_1 :*: e_2) = M_1 (L_1 (R_1 (M_1 (Rec_1 e_1 :×: Rec_1 e_2))))
  from_1 (EVar v)     = M_1 (R_1 (L_1 (M_1 (M_1 (Rec_1 v)))))
  from_1 (Let d e)    = M_1 (R_1 (R_1 (M_1 (Comp_1 (gmap Rec_1 d) :×: Rec_1 e))))
  to_1 (M_1 (L_1 (L_1 (M_1 (K_1 i)))))                  = Const i
  to_1 (M_1 (L_1 (R_1 (M_1 (Rec_1 e_1 :×: Rec_1 e_2))))) = e_1 :*: e_2
  to_1 (M_1 (R_1 (L_1 (M_1 (M_1 (Rec_1 v))))))          = EVar v
  to_1 (M_1 (R_1 (R_1 (M_1 (Comp_1 d :×: Rec_1 e)))))   = Let (gmap unRec_1 d) e
```

We need to use gmap to apply the Rec_1 constructor inside the lists. In this case we could use map instead, but in general we require the first argument of :o: to have a GFunctor instance so we can use gmap (see Section 11.3). In to_1 we need to convert back, this time mapping unRec_1. Note that the cases for EVar use two sequential M_1 constructors, one for constructor and the other for record selector meta-information.

The embedding-projection pair for Var is similar:

```
instance Generic_1 Var where
  type Rep_1 Var = Rep_1^Var
  from_1 (Var x)    = M_1 (L_1 (M_1 (Par_1 x)))
  from_1 (VarL xs)  = M_1 (R_1 (M_1 (Comp_1 (gmap Rec_1 xs))))
  to_1 (M_1 (L_1 (M_1 (Par_1 x))))      = Var x
  to_1 (M_1 (R_1 (M_1 (Comp_1 xs))))    = VarL (gmap unRec_1 xs)
```

Note that composition is used both in the representation for the first argument of constructor Let (of type $[\,\mathsf{Decl}\ \rho\,]$) and in the nested recursion of VarL (of type Var $[\,\rho\,]$). In both cases, we have a recursive occurrence of a parameterised datatype where the parameter is not just the variable ρ. Recall our definition of composition:

```
data (:∘:) φ ψ ρ = Comp1 (φ (ψ ρ))
```

The type ϕ is applied not to ρ, but to the result of applying ψ to ρ. This is why we use :∘: when the recursive argument to a datatype is not ρ, like in [Decl ρ] and Var [ρ]. When it is ρ, we can simply use Rec_1.

We have seen how to represent many features of Haskell datatypes in our approach. We give a detailed discussion of the supported datatypes in Section 11.8.1.

11.3 Generic functions

In this section we show how to define type classes with a default generic implementation.

11.3.1 Generic function definition

Recall function encode from Section 11.2:

```
data Bit = O | I

class Encode α where
  encode :: α → [Bit]
```

We cannot provide instances of Encode for our representation types, as those have kind $\star \to \star$, and Encode expects a parameter of kind $\star$. We therefore define a helper class, this time parameterised over a variable of kind $\star \to \star$:

```
class Encode1 φ where
  encode1 :: φ χ → [Bit]
```

For constructors without arguments we return the empty list, as there is nothing to encode. Meta-information is discarded:

```
instance Encode1 U1 where
  encode1 _ = []
instance (Encode1 φ) ⇒ Encode1 (M1 ι γ φ) where
  encode1 (M1 a) = encode1 a
```

For a value of a sum type we produce a single bit to record the choice. For products we concatenate the encodings of both elements:

```
instance (Encode1 φ, Encode1 ψ) ⇒ Encode1 (φ :+: ψ) where
  encode1 (L1 a)  = O : encode1 a
  encode1 (R1 a)  = I : encode1 a
instance (Encode1 φ, Encode1 ψ) ⇒ Encode1 (φ :×: ψ) where
  encode1 (a :×: b) = encode1 a ++ encode1 b
```

It remains to encode constants. Since constant types have kind $\star$, we resort to Encode:

```
instance (Encode φ) ⇒ Encode₁ (K₁ ι φ) where
  encode₁ (K₁ a) = encode a
```

Note that while the instances for the representation types are given for the Encode_1 class, only the Encode class is exported and available for instantiation. This is because its type is more general, and because we need a two-level approach to deal with recursion: for the K_1 instance, we recursively call encode instead of encode_1. Recall our representation for Exp (simplified and with type synonyms expanded):

```
type Rep₀^Exp = K₁ R Int :+: K₁ R Exp :×: K₁ R Exp
```

Since Int and Exp appear as arguments to K_1, and our instance of Encode_1 for $\text{K}_1\ \iota\ \phi$ requires an instance of Encode ϕ, we need instances of Encode for Int and for Exp. We deal with Int in the next section, and Exp in Section 11.3.3. Finally, note that we do not need Encode_1 instances for Rec_1, Par_1 or :○:. These are only required for generic functions which make use of the Generic_1 class. We will see an example in Section 11.3.4.

11.3.2 Base types

We have to provide the instances of Encode for the base types:

```
instance Encode Int  where encode = ...
instance Encode Char where encode = ...
```

Since Encode is exported, a user can also provide additional base type instances, or ad-hoc instances (types for which the required implementation is different from the derived generic behavior).

11.3.3 Default definition

We miss an instance of Encode for Exp. Instances of generic functions for representable types rely on the embedding-projection pair to convert from/to the type representation and then apply the generic function:

```
encode_Default :: (Generic α, Encode₁ (Rep α)) ⇒ α → [Bit]
encode_Default = encode₁ ∘ from
```

The function $\text{encode}_{\text{Default}}$ is what we want to use as implementation of encode for types with a Generic instance. We want it to be the *default* implementation of encode: if the user does not provide an adhoc implementation, we should use this one. Unfortunately we cannot make it a standard Haskell 98 default, as in the following code:

```
class Encode α where
  encode :: α → [Bit]
  encode = encode_Default
```

This code does not type-check, as the constraint of $\text{encode}_{\text{Default}}$ is not satisfied. Adding the constraint (Generic α, Encode_1 (Rep α)) to the Encode class is not a good solution, because it would prevent giving adhoc instances for types that are not instances of Generic.

What we want is to say that the default method of encode should have a different type signature. This signature applies only to the default method, so it is only used when filling-in an instance that was left empty by the user. We call this a *default signature*:

```
class Encode α where
  encode :: α → [Bit]
  default encode :: (Generic α, Encode₁ (Rep α)) ⇒ α → [Bit]
  encode = encode₁ ∘ from
```

In this way we can give the desired default implementation for encode, together with the correct type signature. We have implemented default signatures as an extension to GHC; it is enabled with the pragma `-XDefaultSignatures`.

Instantiating user datatypes is now a simple matter of giving an instance head. For instance, the instance of Encode for Exp is as follows:

```
instance Encode Exp
```

The instance for lists is similar, only that we need to specify the Encode constraint on the list argument:

```
instance (Encode α) ⇒ Encode [α]
```

Default definitions are handled differently in UHC. We explain these in Section 11.5.

11.3.4 Generic map

In this section we define the generic map function gmap, which implements the Prelude's fmap function. Function gmap requires access to the parameter in the representation type. As before, we export a single class, and use an internal class to define the generic instances:

```
class GFunctor φ where
  gmap :: (ρ → α) → φ ρ → φ α

class GFunctor₁ φ where
  gmap₁ :: (ρ → α) → φ ρ → φ α
```

Unlike in Encode, the type arguments to GFunctor and GFunctor_1 have the same kind, so we do not really need two classes. However, we argue that these are really two different classes. GFunctor is a user-facing class, available for being instantiated at any type ϕ of kind $\star \to \star$. GFunctor_1, on the other hand, is only used by the generic programmer for giving instances for the representation types.

We apply the argument function in the parameter case:

```
instance GFunctor1 Par1 where
  gmap1 f (Par1 a) = Par1 (f a)
```

Unit and constant values do not change, as there is nothing we can map over. We apply gmap_1 recursively to meta-information, sums, and products:

```
instance GFunctor1 U1 where
  gmap1 f U1       = U1
instance GFunctor1 (K1 ι γ) where
  gmap1 f (K1 a)   = K1 a
instance (GFunctor1 φ) ⇒ GFunctor1 (M1 ι γ φ) where
  gmap1 f (M1 a)   = M1 (gmap1 f a)
instance (GFunctor1 φ, GFunctor1 ψ) ⇒ GFunctor1 (φ :+: ψ) where
  gmap1 f (L1 a)   = L1 (gmap1 f a)
  gmap1 f (R1 a)   = R1 (gmap1 f a)
instance (GFunctor1 φ, GFunctor1 ψ) ⇒ GFunctor1 (φ :×: ψ) where
  gmap1 f (a :×: b) = gmap1 f a :×: gmap1 f b
```

If we find a recursive occurrence of a functorial type we call gmap again, to tie the recursive knot:

```
instance (GFunctor φ) ⇒ GFunctor1 (Rec1 φ) where
  gmap1 f (Rec1 a) = Rec1 (gmap f a)
```

The remaining case is composition:

```
instance (GFunctor φ, GFunctor1 ψ) ⇒ GFunctor1 (φ :∘: ψ) where
  gmap1 f (Comp1 x) = Comp1 (gmap (gmap1 f) x)
```

Recall that we require the first argument of :∘: to be a user-defined datatype, and the second to be a representation type. Therefore, we use gmap_1 for the inner mapping (as it will map over a representation type) but gmap for the outer mapping (as it will require an embedding-projection pair). This is the general structure of the instance of :∘: for a generic function.

Finally we define the default method. It is part of the GFunctor class, which we have seen before, but we show the default only now because it depends on gmap_1:

```
default gmap :: (Generic1 φ, GFunctor1 (Rep1 φ)) ⇒ (ρ → α) → φ ρ → φ α
gmap f = to1 ∘ gmap1 f ∘ from1
```

Now GFunctor can be derived for user-defined datatypes. The usual restrictions apply: only types with at least one type parameter and whose last type argument is of kind $\star$ can derive GFunctor. Defining a GFunctor instance for lists is now very simple:

```
instance GFunctor []
```

The instance GFunctor [] also guarantees that we can use [] as the first argument to :∘:, as the embedding-projection pairs for such compositions need to use gmap.

11.3.5 Generic empty

We can also easily express generic producers (functions which produce data). We will illustrate this with function empty, which produces a single value of a given type:

```
class Empty α where
  empty :: α
```

This function is perhaps the simplest generic producer, as it consumes no data. It relies only on the structure of the datatype to produce values. Other examples of generic producers are the methods in Read, the Arbitrary class from `QuickCheck`, and `binary`'s get. As usual, we define an auxiliary type class:

```
class Empty₁ φ where
  empty₁ :: φ χ
```

Most instances of Empty_1 are straightforward:

```
instance Empty₁ U₁ where
  empty₁ = U₁
instance (Empty₁ φ) ⇒ Empty₁ (M₁ ι γ φ) where
  empty₁ = M₁ empty₁
instance (Empty₁ φ, Empty₁ ψ) ⇒ Empty₁ (φ :×: ψ) where
  empty₁ = empty₁ :×: empty₁
instance (Empty φ) ⇒ Empty₁ (K₁ ι φ) where
  empty₁ = K₁ empty
```

For units we can only produce U_1. Meta-information is produced with M_1, and since we encode the meta-information using type classes (instead of using extra arguments to M_1) we do not have to use $\bot$ here. An empty product is the product of empty components, and for K_1 we recursively call empty. The only interesting choice is for the sum type:

```
instance (Empty₁ φ) ⇒ Empty₁ (φ :+: ψ) where
  empty₁ = L₁ empty₁
```

In a sum, we always take the leftmost constructor for the empty value. Since the leftmost constructor might be recursive, function empty might not terminate. More complex implementations can look ahead to spot recursion, or choose alternative constructors after recursive calls, for instance. Note also the similarity between our Empty class and Haskell's Bounded: if we were defining minBound and maxBound generically, we could choose L_1 for minBound and R_1 for maxBound. This way we would preserve the semantics for derived Bounded instances, as defined by Peyton Jones [2003], while at the same time lifting the restrictions on types that can derive Bounded. Alternatively, to keep the Haskell 98 behavior, we could give no instance for :×:, as enumeration types will not have a product in their representations.

The default method (part of the Empty class) simply applies to to empty_1:

```
default empty :: (Generic α, Empty1 (Rep α)) ⇒ α
empty = to empty1
```

Generic instances can now be defined:

```
instance Empty Exp
instance (Empty ρ) ⇒ Empty [ρ]
```

Instances for other types are similar.

11.3.6 Generic show

To illustrate the use of constructor and selector labels, we define the shows function generically:

```
class GShow α where
  gshows :: α → ShowS
  gshow  :: α → String
  gshow x = gshows x ""
```

We define a helper class GShow_1, with gshows_1 as the only method. For each representation type there is an instance of GShow_1. The extra Bool argument will be explained later. Datatype meta-information and sums are ignored. For units we have nothing to show, and for constants we call gshows recursively:

```
class GShow1 φ where
  gshows1 :: Bool → φ χ → ShowS

instance (GShow1 φ) ⇒ GShow1 (D1 γ φ) where
  gshows1 b (M1 a) = gshows1 b a
instance (GShow1 φ, GShow1 ψ) ⇒ GShow1 (φ :+: ψ) where
  gshows1 b (L1 a) = gshows1 b a
  gshows1 b (R1 a) = gshows1 b a
instance GShow1 U1 where
  gshows1 _ U1     = id
instance (GShow φ) ⇒ GShow1 (K1 ι φ) where
  gshows1 _ (K1 a) = gshows a
```

The most interesting instances are for the meta-information of a constructor and a selector. For simplicity, we always place parentheses around a constructor and ignore infix operators. We do display a labeled constructor with record notation. At the constructor level, we use conIsRecord to decide if we print surrounding brackets or not. We use the Bool argument to gshows_1 to encode that we are inside a labeled field, as we will need this for the product case:

```
instance (GShow1 φ, Constructor γ) ⇒ GShow1 (C1 γ φ) where
  gshows1 _ c@ (M1 a) = showString "(" ∘ showString (conName c)
                      ∘ showString " "
                      ∘ wrapR (gshows1 (conIsRecord c) a ∘ showString ")")
    where wrapR :: ShowS → ShowS
          wrapR s | conIsRecord c = showString "{ " ∘ s ∘ showString " }"
          wrapR s | otherwise     = s
```

For a selector, we print its label (as long as it is not empty), followed by an equals sign and the value. In the product, we use the Bool to decide if we print a space (unlabeled constructors) or a comma:

```
instance (GShow1 φ, Selector γ) ⇒ GShow1 (S1 γ φ) where
  gshows1 b s@ (M1 a)
    | null (selName s) = gshows1 b a
    | otherwise        = showString (selName s)
                       ∘ showString " = " ∘ gshows1 b a
instance (GShow1 φ, GShow1 ψ) ⇒ GShow1 (φ :×: ψ) where
  gshows1 b (a :×: c) = gshows1 b a
                      ∘ showString (if b then "," else " ")
                      ∘ gshows1 b c
```

Finally, we show the default method:

```
default gshows :: (Generic α, GShow1 (Rep α)) ⇒ α → ShowS
gshows = gshows1 False ∘ from
```

We have shown how to use meta-information to define a generic show function. If we additionally account for infix constructors and operator precedence for avoiding unnecessary parentheses, we obtain a formal specification of how show behaves on most Haskell 98 datatypes.

11.4 Compiler support

We now describe in detail the required compiler support for our generic deriving mechanism.

We start by defining two predicates on types, isRep0 ϕ and isRep1 ϕ, which hold if ϕ can be made an instance of Generic and Generic$_1$, respectively. The statement isRep0 ϕ holds if ϕ is any of the following:

1. A Haskell 98 datatype without context
2. An empty datatype
3. A type variable of kind $\star$

We also require that for every type ψ that appears as an argument to a constructor of ϕ, isRep0 ψ holds. ϕ cannot use existential quantification, type equalities, or any other extensions.

The statement isRep1 ϕ holds if the following conditions both hold:

1. isRep0 ϕ
2. ϕ is of kind $\star \to \star$ or $k \to \star \to \star$, for any kind k

Note that isRep0 holds for all the types of Section 11.2.4, while isRep1 holds for [], Expr, Decl, and Var.

Furthermore, we define the predicate ground ϕ to determine whether a datatype has type variables. For instance, ground [Int] holds, but ground [α] does not. Finally, we assume the existence of an indexed fresh variable generator fresh p_i^j, which binds p_i^j to a unique fresh variable.

For the remainder of this section, we consider a user-defined datatype

$$\begin{aligned} \textbf{data}\ D\ \alpha_1\ \ldots\ \alpha_n\ = \ & Con_1\ \{l_1^1 :: p_1^1,\ \ldots, l_1^{o_1} :: p_1^{o_1}\} \\ & \vdots \\ & \mid\ Con_m\ \{l_m^1 :: p_m^1,\ \ldots, l_m^{o_m} :: p_m^{o_m}\} \end{aligned}$$

with n type parameters, m constructors and possibly labeled parameter l_i^j of type p_i^j at position j of constructor Con_i.

11.4.1 Type representation (kind $\star$)

Figure 11.1 shows the generation of type representations for a datatype D satisfying isRep0 D. We generate a number of empty datatypes which we use in the meta-information: one for the datatype, one for each constructor, and one for each argument to a constructor that has a selector field.

The type representation is a type synonym (Rep_0^D) with as many type variables as D. It is a wrapped sum of wrapped products: the wrapping encodes the meta-information. We wrap all arguments to constructors, even if the constructor is not a record. However, when the argument does not have a selector field we encode it with S NoSelector. Unlike the generated empty datatypes, NoSelector is exported to the user, who can match on it when defining generic functions. Since we use a balanced sum (resp. product) encoding, a generic function can use the meta-information to find out when the sum (resp. product) structure ends, which is when we reach C_1 (resp. S_1). Each argument is tagged with Par_0 if it is one of the type variables, or Rec_0 if it is anything else (type application or a concrete datatype).

11.4.2 Generic instance

Instantiation of the Generic class (introduced in Section 11.2) is defined in Figure 11.2. The patterns of the from function are the constructors of the datatype applied to fresh

$$\textbf{type } \mathrm{Rep}_0^{\$D}\ \alpha_1\ \ldots\ \alpha_n\ =\ D_1\ \$D\ (\textstyle\sum_{i=1}^{m}(C_1\ \$Con_i\ (\prod_{j=1}^{o_i}(S_1\ (\mathrm{sel}\ \$L_i^j)\ (\mathrm{arg}\ p_i^j)))))$$

$$\begin{array}{ll}
 & \mathrm{sel}\ \$L_i^j \mid l_i^j \text{ is defined} = \$L_i^j \\
 & \qquad\quad \mid \text{otherwise} = \mathrm{NoSelector} \\
\textbf{data } \$D & \\
\textbf{data } \$Con_1 & \textstyle\sum_{i=1}^{n} x \mid n \equiv 0 = V_1 \\
 & \qquad\quad \mid n \equiv 1 = x \\
\vdots & \qquad\quad \mid \text{otherwise} = \textstyle\sum_{i=1}^{m} x\ :+:\ \sum_{i=1}^{n-m} x\ \textbf{where}\ m = \lfloor n/2 \rfloor \\
\textbf{data } \$Con_m & \\
 & \textstyle\prod_{i=1}^{n} x \mid n \equiv 0 = U_1 \\
\textbf{data } \$L_1^1 & \qquad\quad \mid n \equiv 1 = x \\
 & \qquad\quad \mid \text{otherwise} = \textstyle\prod_{i=1}^{m} x\ :\times:\ \prod_{i=1}^{n-m} x\ \textbf{where}\ m = \lfloor n/2 \rfloor \\
\vdots & \\
\textbf{data } \$L_m^{o_m} & \mathrm{arg}\ p_i^j \mid \exists_{k \in \{1\,..n\}} : p_i^j \equiv \alpha_k = \mathrm{Par}_0\ p_i^j \\
 & \qquad\quad \mid \text{otherwise} = \mathrm{Rec}_0\ p_i^j
\end{array}$$

Figure 11.1: Code generation for the type representation (kind $\star$)

variables. The same patterns become expressions in function to. The patterns of to are also the same as the expressions of from, and they represent the different values of a balanced sum of balanced products, properly wrapped to account for the meta-information. Note that, for Generic, the functions tuple and wrap do not behave differently depending on whether we are in from or to, so for these declarations the dir argument is not needed. Similarly, the wrap function could have been inlined. These definitions will be refined in Section 11.4.4.

11.4.3 Type representation (kind $\star \to \star$)

Figure 11.3 shows the type representation of type constructors. We keep the sum-of-products structure and meta-information unchanged. At the arguments, however, we can use Par_0, Par_1, Rec_0, Rec_1, or composition. We use Par_1 for the type variable α, and Par_0 for other type variables of kind $\star$. A recursive occurrence of a type containing α_n is marked with Rec_1. A recursive occurrence of a type with no type variables is marked with Rec_0, as there is no variable to abstract from. Finally, for a recursive occurrence of a type which contains anything other than α_n we use composition, and recursively analyse the contained type.

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Generic}\ (\mathsf{D}\ \alpha_1\ \ldots\ \alpha_n)\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{Rep}\ (\mathsf{D}\ \alpha_1\ \ldots\ \alpha_n) = \mathsf{Rep}_0^{\mathsf{D}}\ \alpha_1\ \ldots\ \alpha_n \\
\quad \mathsf{from}\quad \mathsf{pat}_1^{\mathsf{from}} = \mathsf{exp}_1^{\mathsf{from}} \\
\qquad\qquad \vdots \\
\quad \mathsf{from}\quad \mathsf{pat}_m^{\mathsf{from}} = \mathsf{exp}_m^{\mathsf{from}} \\
\quad \mathsf{to}\quad \mathsf{pat}_1^{\mathsf{to}} = \mathsf{exp}_1^{\mathsf{to}} \\
\qquad\qquad \vdots \\
\quad \mathsf{to}\quad \mathsf{pat}_m^{\mathsf{to}} = \mathsf{exp}_m^{\mathsf{to}}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{exp}_i^{\mathsf{to}} = \mathsf{pat}_i^{\mathsf{from}} = \mathsf{Con}_i\ (\mathsf{fresh}\ p_i^1)\ \ldots\ (\mathsf{fresh}\ p_i^{o_i}) \\
\mathsf{exp}_i^{\mathsf{from}} = \mathsf{pat}_i^{\mathsf{to}} = \mathsf{M}_1\ (\mathsf{inj}_{i,m}\ (\mathsf{M}_1\ (\mathsf{tuple}_i^{-}\ p_i^1\ \ldots\ p_i^{o_i})))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{inj}_{i,m}\ x \mid m \equiv 0 = \bot \\
\qquad\quad \mid m \equiv 1 = x \\
\qquad\quad \mid i \leqslant m' = \mathsf{L}_1\ (\mathsf{inj}_{i,m'}\ x) \\
\qquad\quad \mid i > m' = \mathsf{R}_1\ (\mathsf{inj}_{i',m\,-\,m'}\ x) \\
\quad \textbf{where}\ m' = \lfloor m/2 \rfloor \\
\qquad\quad\ \ i' = \lfloor i/2 \rfloor
\end{array}
$$

$$
\begin{array}{l}
\mathsf{tuple}_i^{\mathsf{dir}}\ p_i^j\ \ldots\ p_i^{o_i} \mid o_i \equiv 0 = \mathsf{M}_1\ \mathsf{U}_1 \\
\qquad\qquad\qquad\qquad\quad \mid o_i \equiv j = \mathsf{M}_1\ (\mathsf{wrap}^{\mathsf{dir}}\ (\mathsf{fresh}\ p_i^j)) \\
\qquad\qquad\qquad\qquad\quad \mid \mathsf{otherwise} = (\mathsf{tuple}_i^{\mathsf{dir}}\ p_i^1\ \ldots\ p_i^k)\ :\times:\ (\mathsf{tuple}_i^{\mathsf{dir}}\ p_i^{k+1}\ \ldots\ p_i^m) \\
\quad \textbf{where}\ k = \lfloor (o_i\ /2) \rfloor
\end{array}
$$

$$
\mathsf{wrap}^{\mathsf{dir}}\ p = \mathsf{K}_1\ p
$$

Figure 11.2: Code generation for the Generic instance

11.4.4 Generic$_1$ instance

The definition of the embedding-projection pair for kind $\star \to \star$ datatypes, shown in Figure 11.4, reflects the more complicated type representation. The patterns are unchanged. However, the expressions in to$_1$ need some additional unwrapping. This is encoded in var and unwC: an application to a type variable other than α_n has been encoded as a composition, so we need to unwrap the elements of the contained type. We use gmap for this purpose: since we require isRep1 ϕ, we know that we can use gmap (see Section 11.3.4). The user should always derive GFunctor for container types, as these can appear to the left of a composition.

Unwrapping is dual to wrapping: we use Par$_1$ for the type parameter α_n, Rec$_1$ for

$$\textbf{type}\ \mathsf{Rep}^{\mathsf{D}}_1\ \alpha_1\ \ldots\ \alpha_{n-1}\ =\ \mathsf{D}_1\ \$\mathsf{D}\ (\textstyle\sum_{i=1}^{m}\ (\mathsf{C}_1\ \$\mathsf{Con}_i\ (\prod_{j=1}^{o_i}\ (\mathsf{S}_1\ (\mathsf{sel}\ \$\mathsf{L}^j_i)\ (\mathsf{arg}\ \mathsf{p}^j_i)))))$$

sel $\$\mathsf{L}^j_i$, $\sum_{i=1}^{m} x$, and $\prod_{j=1}^{n} x$ as in Figure 11.1.

$$\begin{array}{rlll}
\mathsf{arg}\ \mathsf{p}^j_i & \mid\ \exists_{k\,\in\,\{1\,..n-1\}}\ :\ \mathsf{p}^j_i\ \equiv\ \alpha_k & = & \mathsf{Par}_0\ \mathsf{p}^j_i \\
& \mid\ \mathsf{p}^j_i\ \equiv\ \alpha_n & = & \mathsf{Par}_1 \\
& \mid\ \mathsf{p}^j_i\ \equiv\ \phi\ \alpha_n\ \wedge\ \mathsf{isRep1}\ \mathsf{f}) & = & \mathsf{Rec}_1\ \mathsf{p}^j_i \\
& \mid\ \mathsf{p}^j_i\ \equiv\ \phi\ \beta\ \ \wedge\ \mathsf{isRep1}\ \phi\ \wedge\ \neg\ \mathsf{ground}\ \beta & = & \phi\ :\circ:\ \mathsf{arg}\ \beta \\
& \mid\ \mathsf{otherwise} & = & \mathsf{Rec}_0\ \mathsf{p}^j_i
\end{array}$$

Figure 11.3: Code generation for the type representation (kind $\star \to \star$)

containers of α_n, K_1 for other type parameters and ground types, and composition for application to types other than α_n. In to_1 we generate only Comp_1 applied to a fresh variable, as this is a pattern; the necessary unwrapping of the contained elements is performed in the right-hand side expression. In from_1 the contained elements are tagged properly: this is performed by wC_α.

11.4.5 Meta-information

We generate three meta-information instances. For datatypes, we generate:

```
instance Datatype $D where
  moduleName   _ = mName
  datatypeName _ = dName
```

Here, dName is a String with the unqualified name of datatype D, and mName is a String with the name of the module in which D is defined.

For constructors, we generate:

```
instance Constructor $Con_i where
    conName     _ = name
  {conFixity    _ = fixity}
  {conIsRecord  _ = True}
```

Here, $i \in \{1..m\}$, and name is the unqualified name of constructor Con_i. The braces around conFixity indicate that this method is only defined if Con_i is an infix constructor. In that case, fixity is Infix assoc prio, where prio is an integer denoting the priority of Con_i, and assoc is one of LeftAssociative, RightAssociative, or NotAssociative. These are derived from the declaration of Con_i as an infix constructor. The braces around conIsRecord indicate that this method is only defined if Con_i uses record notation.

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Generic}_1\ (\mathsf{D}\ \alpha_1\ \ldots\ \alpha_{n-1})\ \textbf{where} \\
\quad \textbf{type}\ \mathsf{Rep}_1\ (\mathsf{D}\ \alpha_1\ \ldots\ \alpha_{n-1}) = \mathsf{Rep}_1^{\mathsf{D}}\ \alpha_1\ \ldots\ \alpha_{n-1} \\
\quad \mathsf{from}_1\ \mathsf{pat}_1^{\mathsf{from}} = \mathsf{exp}_1^{\mathsf{from}} \\
\qquad\qquad \vdots \\
\quad \mathsf{from}_1\ \mathsf{pat}_m^{\mathsf{from}} = \mathsf{exp}_m^{\mathsf{from}} \\
\quad \mathsf{to}_1\ \ \mathsf{pat}_1^{\mathsf{to}} = \mathsf{exp}_1^{\mathsf{to}} \\
\qquad\qquad \vdots \\
\quad \mathsf{to}_1\ \ \mathsf{pat}_m^{\mathsf{to}} = \mathsf{exp}_m^{\mathsf{to}}
\end{array}
$$

$\mathsf{pat}_i^{\mathsf{dir}}$, $\mathsf{exp}_i^{\mathsf{from}}$, $\mathsf{inj}_{i,m}$ x, and $\mathsf{tuple}_i^{\mathsf{dir}}\ p_1\ \ldots\ p_m$ as in Figure 11.2 (but using the new $\mathsf{wrap}^{\mathsf{dir}}$ x).

$$
\mathsf{exp}_i^{\mathsf{to}} = \mathsf{Con}_i\ (\mathsf{var}\ p_i^1)\ \ldots\ (\mathsf{var}\ p_i^{o_i})
$$

$$
\begin{array}{lll}
\mathsf{var}\ p_i^j & \mid p_i^j \equiv \phi\ \alpha \wedge \alpha \not\equiv \alpha_n \wedge \mathsf{isRep1}\ \phi & = \mathsf{gmap}\ \mathsf{unwC}_\alpha\ (\mathsf{fresh}\ p_i^j) \\
& \mid \mathsf{otherwise} & = (\mathsf{fresh}\ p_i^j)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{wrap}^{\mathsf{dir}}\ p_i^j & \mid p_i^j \equiv \alpha_n & = \mathsf{Par}_1\ (\mathsf{fresh}\ p_i^j) \\
& \mid p_i^j \equiv \phi\ \alpha_n \wedge \mathsf{isRep1}\ \phi & = \mathsf{Rec}_1\ (\mathsf{fresh}\ p_i^j) \\
& \mid \exists_{k \in \{1\,..n\}} : p_i^j \equiv \alpha_k & = \mathsf{K}_1\ (\mathsf{fresh}\ p_i^j) \\
& \mid p_i^j \equiv \phi\ \alpha \wedge \neg\, \mathsf{isRep1}\ \phi & = \mathsf{K}_1\ (\mathsf{fresh}\ p_i^j) \\
& \mid p_i^j \equiv \phi\ \alpha \wedge \mathsf{dir} \equiv \mathsf{from} & = \mathsf{Comp}_1\ (\mathsf{gmap}\ \mathsf{wC}_\alpha\ (\mathsf{fresh}\ p_i^j)) \\
& \mid \mathsf{otherwise} & = \mathsf{Comp}_1\ (\mathsf{fresh}\ p_i^j)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{unwC}_\alpha & \mid \alpha \equiv \alpha_n & = \mathsf{unPar}_1 \\
& \mid \alpha \equiv \phi\ \alpha_n \wedge \mathsf{isRep1}\ \phi & = \mathsf{unRec}_1 \\
& \mid \alpha \equiv \phi\ \beta \wedge \mathsf{ground}\ \beta & = \mathsf{unRec}_0 \\
& \mid \alpha \equiv \phi\ \beta \wedge \mathsf{isRep1}\ \phi & = \mathsf{gmap}\ \mathsf{unwC}_\beta \circ \mathsf{unComp1}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{wC}_\alpha & \mid \alpha \equiv \alpha_n & = \mathsf{Par}_1 \\
& \mid \mathsf{ground}\ \alpha & = \mathsf{K}_1 \\
& \mid \alpha \equiv \phi\ \alpha_n \wedge \mathsf{isRep1}\ \phi & = \mathsf{Rec}_1 \\
& \mid \alpha \equiv \phi\ \beta \wedge \mathsf{isRep1}\ \phi & = \mathsf{Comp}_1 \circ (\mathsf{gmap}\ \mathsf{wC}_\beta)
\end{array}
$$

Figure 11.4: Code generation for the $\mathsf{Generic}_1$ instance

For all $i \in \{1..m\}$, if a selector label l_i^j is defined, we generate:

```
instance Selector $L_i^j where
  selName _ = l_i^j
```

Here, $j \in \{1..o_i\}$. If the selector label is not defined then the representation type uses NoSelector, which also has a Selector instance:

```
data NoSelector
instance Selector NoSelector where
  selName _ = ""
```

11.5 Implementation in UHC

So far we have presented the implementation of our technique in GHC. An earlier version of our work was implemented in UHC. In this section we detail the differences between the two implementations.

The main reason behind the differences is UHC's lack of support for advanced type-level programming features, such as type families or functional dependencies. This makes the task of encoding the isomorphism between a datatype and its representation more cumbersome. On the other hand, the UHC implementation reveals that a generic programming framework can still be integrated into a compiler even without many extensions beyond Haskell 98; the only significant extension we need is support for multi-parameter type classes.

11.5.1 Datatype representation

In UHC, we use a multi-parameter type class for embedding-projection pairs:

```
class Representable0 α τ where
  from0 :: α → τ χ
  to0   :: τ χ → α

class Representable1 φ τ where
  from1 :: φ ρ → τ ρ
  to1   :: τ ρ → φ ρ
```

The variable τ encodes the representation type. It is uniquely determined by α (and ϕ), but we avoid the use of functional dependencies. Note also the different naming of the classes and their methods.

Instantiation remains mostly unchanged, apart from the positioning of the representation type. See, for example, the instantiation of lists (using the types defined in Section 11.2.4):

```
instance Representable0 [ ρ ] (Rep0List ρ) where
  from0 []       = M1 (L1 (M1 U1))
  from0 (h : t)  = M1 (R1 (M1 (K1 h :×: K1 t)))
  to0 (M1 (L1 (M1 U1)))                 = []
  to0 (M1 (R1 (M1 (K1 h :×: K1 t))))    = h : t

instance Representable1 [] Rep1List where
  from1 []       = M1 (L1 (M1 U1))
  from1 (h : t)  = M1 (R1 (M1 (Par1 h :×: Rec1 t)))
  to1 (M1 (L1 (M1 U1)))                 = []
  to1 (M1 (R1 (M1 (Par1 h :×: Rec1 t))))  = h : t
```

In particular, the code for the methods is exactly the same.

11.5.2 Specifying default methods

UHC does not have support for default signatures, so we cannot use those. Instead, we define the default definition of each generic function separately, and then tie it together with its class using a pragma. For instance, for Encode the generic function developer has to define:

```
encodeDefault :: (Representable0 α τ, Encode1 τ) ⇒ τ χ → α → [Bit]
encodeDefault rep x = encode1 ((from0 x) `asTypeOf` rep)
```

Because we do not use functional dependencies, we have to pass the representation type explicitly to the function $\text{encode}_{\text{Default}}$. This function then uses the representation type to coerce the result type of from with asTypeOf.

The instances of Encode for Exp and [] are as follows:

```
instance Encode Exp where
  encode = encodeDefault (⊥ :: Rep0Exp χ)
instance (Encode ρ) ⇒ Encode [ ρ ] where
  encode = encodeDefault (⊥ :: Rep0List ρ χ)
```

Note, however, that the instance for [] requires scoped type variables to type-check. We can avoid the need for scoped type variables if we create an auxiliary local function $\text{encode}_{[]}$ with the same type and behavior of $\text{encode}_{\text{Default}}$:

```
instance (Encode ρ) ⇒ Encode [ ρ ] where
  encode = encode[] ⊥ where
    encode[] :: (Encode ρ) ⇒ Rep0List ρ χ → [ ρ ] → [Bit]
    encode[] = encodeDefault
```

Here, the local function $\text{encode}_{[]}$ encodes in its type the correspondence between the type [ρ] and its representation $\text{Rep}_0^{\text{List}}\ \rho$. Its type signature is required, but can easily

be obtained from the type of $\mathsf{encode}_{\mathsf{Default}}$ by replacing the type variables α and τ with the concrete types for this instance.

These instances are considerably more complicated than the empty instances we use in GHC, but they are automatically generated by UHC. However, we still need a mechanism to tie the default definitions to their respective classes. We use a pragma for this purpose:

$$\{-\#\ \mathsf{DERIVABLE}\ \mathsf{Encode}\ \mathsf{encode}\ \mathsf{encode}_{\mathsf{Default}}\ \#-\}$$

This pragma takes three arguments, which represent (respectively):

1. The class which we are defining as derivable
2. The method of the class which is generic (and therefore needs a default definition)
3. The name of the function which serves as a default definition

Since a class can have multiple generic methods, multiple pragmas can be used for this purpose.

11.5.3 Instantiating generic functions

After the generic programmer has defined $\mathsf{encode}_{\mathsf{Default}}$ and given the pragma, the user still has to specify that a specific datatype should get a generic instance. Unlike in GHC, in UHC we do use **deriving** for this purpose. So, given a generic function f that is a method of the type class F, and for every datatype D that derives F with type arguments $\alpha_1 \ \ldots \ \alpha_n$ and associated representation type $\mathsf{Rep}_0^{\mathsf{D}}\ \alpha_1 \ \ldots \ \alpha_n\ \chi$, the compiler generates:

$$\begin{array}{l}\textbf{instance}\ \mathsf{Ctx} \Rightarrow \mathsf{F}\ (\mathsf{D}\ \alpha_1 \ \ldots \ \alpha_n)\ \textbf{where} \\ \quad \mathsf{f} = \mathsf{f}_{\mathsf{D}}\ \bot \\ \quad\quad \textbf{where}\ \mathsf{f}_{\mathsf{D}} :: \mathsf{Ctx} \Rightarrow \mathsf{Rep}_0^{\mathsf{D}}\ \alpha_1 \ \ldots \ \alpha_n\ \chi \rightarrow \beta \\ \quad\quad\quad\quad \mathsf{f}_{\mathsf{D}} = \mathsf{f}_{\mathsf{Default}}\end{array}$$

The type β is the type of f specialised to D, and χ is a fresh type variable. The context Ctx is the same in the instance head and in function f_{D}. The exact context generated depends on the way the user specifies the deriving. If **deriving** F is attached to the datatype, we generate a context $\mathsf{F}\ \overrightarrow{\alpha_1}, \ldots, \mathsf{F}\ \overrightarrow{\alpha_n}$, where $\overrightarrow{\alpha}$ is the variable α applied to enough fresh type variables to achieve full saturation. This approach gives the correct behavior for Haskell 98 derivable classes like Show.

In general, however, this is not correct: we cannot assume that we require $\mathsf{F}\ \alpha_i$ for all $i \in \{1..n\}$: some generic functions do not require any constraints because they do not recurse into subterms. Even worse, we might require constraints other than these, as a generic function can use other functions, for instance.

To avoid these problems we can use the standalone deriving extension, which is supported by UHC. The important difference between standalone deriving and standard deriving is that the user supplies the context directly:

deriving instance (Ctx) ⇒ F (D α_1 ... α_n)

Then we can simply use this context for the instance.

The use of the **deriving** clause left us in a unsatisfactory situation. While simply attaching **deriving** F to a datatype declaration is a simple and concise syntax, it does not always work, because in some cases the compiler cannot infer the right context, and as such the user is forced to supply a standalone deriving instance. It does, however, follow the behavior of **deriving** for the Haskell 98 derivable classes, as both UHC and GHC cannot always infer the right context (for instance for nested datatypes). In contrast, our use of default method signatures and empty instance declarations in GHC leads to a more consistent instantiation strategy.

11.6 Alternatives

We have described how to implement a **deriving** mechanism that can be used to specify many datatype-generic functions in Haskell. There are other alternatives, of varying complexity and type-safety.

11.6.1 Pre-processors

The simplest, most powerful, but least type safe alternative to our approach is to implement **deriving** by pre-processing the source file(s), analysing the datatypes definitions and generating the required instances with a tool such as DrIFT [Winstanley and Meacham, 2008]. This requires no work from the compiler writer, but does not simplify the task of adding new derivable classes, as programming by generating strings is not very convenient.

Staged meta-programming (Section 2.1) lies in between a pre-processor and an embedded datatype-generic representation. GHC supports Template Haskell [Sheard and Peyton Jones, 2002], which has become a standard tool for obtaining reflection in Haskell. While Template Haskell provides possibly more flexibility than the purely library-based approach we describe, it imposes a significant hurdle on the compiler writer, who has not only to implement a language for staged programming (if one does not yet exist for the compiler, like in UHC), but also to keep this complex component up-to-date with the rest of the compiler, as it evolves. As evidence of this, Template Haskell support for GADTs and type families only arrived much later than the features themselves. Also, for the derivable class writer, using Template Haskell is more cumbersome and error-prone than writing a datatype-generic definition in Haskell itself.

For these reasons we think that our library-based approach, while having some limitations, offers a good balance between expressive power, type safety, and the amount of implementation effort required.

11.6.2 Library choice

Another design choice we made was in the specific library approach to use. We have decided not to use any of the existing libraries but instead to develop yet another one. However, our library is merely a variant of existing libraries, from which it borrows many ideas. We see our representation as a mixture between `polyp` and `instant-generics`. We share the functorial view with `polyp`, but we abstract from a single type parameter, and not from the recursive occurrence. Our library can also be seen as `instant-generics` extended with a single type parameter. Having one parameter allows us to deal with composition effectively, and we do not duplicate the representation for types without parameters.

11.7 Related work

Clean [Alimarine and Plasmeijer, 2001] has also integrated generic programming directly in the language. We think our approach is more lightweight: we express our generic functions almost entirely in Haskell and require only one small syntactic extension. On the other hand, the approach taken in Clean allows defining generic functions with polykinded types [Hinze, 2002], which means that the function bimap (see Section 11.2.1), for instance, can be defined. Not all Clean datatypes are supported: quantified types, for example, cannot derive generic functions. Our approach does not support all features of Haskell datatypes, but most common datatypes and generic functions are supported.

An extension for derivable type classes similar to ours has been developed by Hinze and Peyton Jones [2001] in GHC. As in Clean, this extension requires special syntax for defining generic functions, which makes it harder to implement and maintain. In contrast, generic functions written in our approach are portable across different compilers. Furthermore, Hinze and Peyton Jones's approach cannot express functions such as gmap, as their type representation does not abstract over type variables. Since GHC version 7.2.1, support for derivable type classes has been dropped, and replaced by the extension we describe in this chapter.

Rodriguez Yakushev et al. [2008] give criteria for comparing generic programming libraries. These criteria consider the library's use of types, and its expressiveness and usability. Regarding types, our library scores very well: we can represent regular, higher-kinded, nested, and mutually recursive datatypes. We can also express subuniverses: generic functions are only applicable to types that derive the corresponding class. We only lack the ability to represent nested higher-kinded datatypes, as our representation abstracts only over a parameter of kind $\star$.

Regarding expressiveness, our library scores good for most criteria: we can abstract over type constructors, give ad-hoc definitions for datatypes, our approach is extensible, supports multiple generic arguments, represents the constructor names and can express consumers, transformers, and producers. We cannot express gmapQ in our approach, but our generic functions are still first-class: we can call generic map with generic show as argument, for instance. Ad-hoc definitions for constructors would be of the form:

```
instance Encode Exp where
  encode (Plus e1 e2) = ...   -- code specific to this constructor
  encode x            = encode_Default
```

Regarding usability, our approach supports separate compilation, is highly portable, has automatic generation of its two representations, requires minimal work to instantiate and define a generic function, is implemented in a compiler and is easy to use. We have not benchmarked our library in UHC. As for GHC, we discuss performance matters of a similar approach in detail in Chapter 9.

11.8 Future work

Our solution is applicable to a wide range of datatypes and can express many generic functions. However, some limitations still remain, and many improvements are possible. In this section we outline some possible directions for future research.

11.8.1 Supported datatypes

Our examples in Section 11.2 show that we can represent many common forms of datatypes. We believe that we can represent all of the Haskell 98 standard datatypes in Generic, except for constrained datatypes. We could easily support constrained datatypes by propagating the constraints to the generic instances.

Regarding Generic_1, we can represent many, but not all datatypes. Consider a nested datatype for representing balanced trees:

```
data Perfect ρ = Node ρ | Perfect (Perfect (ρ , ρ))
```

We cannot give a representation of kind $\star \to \star$ for Perfect, since for the Perfect constructor we would need something like Perfect :∘: Rec_1 ((,) ρ). However, the type variable ρ is no longer available, because we abstract from it. This limitation is caused by the fact that we abstract over a single type parameter. The approach taken by Hesselink [2009] is more general and fits closely with our approach, but it is not clear if it is feasible without advanced language extensions.

Note that for this particular case we could use a datatype which pairs elements of a single type:

```
data Pair ρ = Pair ρ ρ
```

The representation for the Perfect constructor could then be Perfect :∘: Rec_1 Pair.

11.8.2 Generic functions

The representation types we propose limit the kind of generic functions we can define. We can express the Haskell 98 standard derivable classes Eq, Ord, Enum, Bounded, Show, and Read, even lifting some of the restrictions imposed on the Enum and Bounded

instances. All of these are expressible for Generic types. Using Generic_1, we can implement Functor, as the parameter of the Functor class is of kind $\star \to \star$. The same holds for Foldable and Traversable. For Typeable we can express Typeable_0 and Typeable_1.

On the other hand, the Data class has very complex generic functions which cannot be expressed with our representation. Function gfoldl, for instance, requires access to the original datatype constructor, something we cannot do with the current representation. We plan to explore if and how we can change our representation to allow us to express more generic functions, also in the light of the conclusions of our formal comparison of Chapter 8.

11.9 Conclusion

We have shown how generic programming can be better integrated in Haskell by revisiting the **deriving** mechanism. All Haskell 98 derivable type classes can be expressed as generic functions in our library, with the advantage of becoming easily readable and portable. Additionally, many other type classes, such as Functor and Foldable, can be defined generically. Adding new classes with generic methods can now be done by generic programmers in regular Haskell; previously, this would be the compiler developer's task, and would be done using code generation, which is more error-prone and verbose.

We have implemented our solution in both UHC and GHC and invite users to experiment with this feature. We hope our work paves the way for a redefinition of the behavior of derived instances for Haskell Prime [Wallace et al., 2007].

CHAPTER 12

Epilogue

This thesis is about abstraction. More concretely, it is about one form of abstraction in programming languages: generic programming. Abstraction is "a good thing": it allows the programmer to focus on the essential parts of the problem, and promotes code reuse by making fewer assumptions on underlying data structure.

Yet, abstraction is hard. It is easier to explain to a child why 2 + 2 is 4 than to explain the addition of any two arbitrary numbers. It is easier for a programmer to define a function that works on a single datatype than to define a function that works on a large class of datatypes.

However hard generic programming might be, its benefits are enormous. Being forced to define functionality for each datatype, and to adapt this functionality whenever a datatype is changed, leads programmers into avoiding the use of many datatypes. A single, universal type, that can accommodate many different types of structure, each with different invariants, might be preferred to having to deal with a number of more specific datatypes. This can quickly lead to a programming style where many advantages of a static type system are compromised, since types stop being informative and diverse. On the other hand, being free from adapting functionality to each datatype leads to explorative use of the type system; using many different datatypes is not a burden, since much functionality comes "for free". We have seen in a non-trivial practical application of generic programming that this freedom enables faster and more productive development, especially during initial development and prototyping [Magalhães and De Haas, 2011].

We want to help preventing programmers from reducing type diversity in their code, as we believe this cripples program development. Unfortunately, many programmers are not even aware that conflating different types, each with detailed structure and invariants, under a single "umbrella type", with loose structure and semantics, is a bad thing. After all, it is a way of increasing abstraction; however, it also increases the chances for things

to go wrong, and for subtle bugs caused by unsatisfied data invariants to creep in. The only solution for this is to increase programmers' awareness of the alternative: generic programming. Driving adoption of a programming paradigm requires a lot of distributed effort, and involves tasks such as language design, compiler development or adaptation, case studies, development of large and realistic examples, and inclusion in educational curricula.

We have contributed to all of these tasks, in one way or another. We started by introducing generic programming as an abstraction mechanism, and then toured through several different approaches to generic programming, gaining knowledge about the power, usefulness, and limitations of each approach by modelling and formally relating the approaches. This is an essential aspect of *understanding* generic programming, and answers the first research question. With this knowledge, we took off to the practical realm, and addressed limitations that often prevent adoption of generic programming: added more datatype support, improved runtime efficiency, and showed how to integrate it natively in the language for better usability. These are all ways to improve the *practical* application of generic programming, and address the second research question. We have also developed case studies and applications of generic programming [Jeuring et al., 2010, Magalhães and De Haas, 2011], but have not included them in this thesis. At the same time, we have taken part in the educational efforts at Utrecht University, teaching students about generic programming at the graduate level. These efforts prove fruitful, as previous students have been using generic programming successfully in a number of startup companies, such as Well-Typed (`http://well-typed.com/`), Vector Fabrics (`http://vectorfabrics.com/`), and Silk (`http://silkapp.com/`).

We believe it is safe to say that Haskell, and consequently wider adoption of generic programming, is at a turning point. With multicore processors becoming commonplace in computers, interest in parallel computing has surged. Traditional imperative programming languages provide very low-level support for parallelism, while Haskell has great potential for providing simple and mostly automated parallelisation [Peyton Jones and Singh, 2009]. In particular, automated data parallelism [Peyton Jones et al., 2008] relies crucially on generic programming to derive vectorised versions of datatypes automatically. The growing evidence of Haskell as a practical programming language for parallel applications, together with the maturity of its libraries and tools through the Haskell Platform effort (`http://hackage.haskell.org/platform/`), will bring the language to the attention of a much wider audience.

Large scale adoption of generic programming will bring a number of new challenges to the field. So far, most approaches have been experimental and research-oriented, poorly documented, and rarely ever supported for a long time. A generic programming approach suited for industrial adoption will need to be not only fast and easy to use, but also reliable, well documented, and maintained, keeping a stable interface. Our work of integrating a generic programming library in the compiler (Chapter 11) provides a good basis for an "industrial-strength" approach: it is easily available because it comes with the compiler, it is fast, easy to use, and can be updated while keeping a stable interface because it is supported directly in the language.

The increased popularity of Haskell does not mean that future efforts in generic

programming will be limited to engineering issues. The advent of practical dependently-typed programming languages brings new opportunities for generic programming, such as representation of value-dependent datatypes, instantiation of generic functions to these datatypes, and generic proofs, to name a few. Dependently-typed programming certainly allows for increased abstraction, and the higher complexity of the type system means that more information is available at the type level; this translates to increased opportunities for generic programming by exploiting the structure of types.

We look forward to the exciting challenges the future will bring to generic programming as a programming practice. Generic programming is not only desirable but also necessary for the development of high quality statically-typed functional programs, and increasingly complex software development requires increasingly abstract programming techniques. As such, we remain committed to facilitating the use of generic programming and evaluating its success in improving the quality of programs, for a future where less is more: less code, but more reliable, simple, and correct code.

Bibliography

Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers*, EPTCS 43, July 2010. doi:10.4204/EPTCS.43.2.

Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In *The 13th International Workshop on the Implementation of Functional Languages*, volume 2312 of *LNCS*, pages 168–185. Springer, 2001. doi:10.1007/3-540-46028-4_11.

Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference*, volume 3125 of *LNCS*, pages 16–31. Springer, 2004. doi:10.1007/978-3-540-27764-4_3.

Artem Alimarine and Sjaak Smetsers. Improved fusion for optimizing generics. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium*, volume 3350 of *LNCS*, pages 203–218. Springer, 2005. doi:10.1007/978-3-540-30557-6_16.

Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 209–257. Springer-Verlag, 2007. ISBN 3-540-76785-1, 978-3-540-76785-5.

Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1999. ISBN 9780201309560.

Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 69–79. ACM, 2004. doi:10.1145/1017472.1017485.

Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In Doaitse S. Swierstra, Pedro R. Henriques, and José Nuno Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608, pages 28–115. Springer-Verlag, 1999. doi:10.1007/10704973_2.

Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction: 4th International Conference*, volume 1422 of *LNCS*, pages 52–67. Springer, 1998.

Richard Bird and Oege de Moor. *Algebra of programming*. Prentice Hall, 1997. ISBN 9780135072455.

Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Add generic types to the Java programming language, April 2001. URL `http://www.jcp.org/en/jsr/detail?id=014`. JSR14.

Neil C.C. Brown and Adam T. Sampson. Alloy: fast generic transformations for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 105–116. ACM, 2009. doi:10.1145/1596638.1596652.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy, 2009. Available at `http://www.cse.unsw.edu.au/~chak/papers/CDL09.html`.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 3–14. ACM, 2010. doi:10.1145/1863543.1863547.

James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM, 2002. doi:10.1145/581690.581698.

Adam Chlipala. *Certified Programming with Dependent Types*, chapter 4. Available at `http://adam.chlipala.net/cpdt/`, 2012.

Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Jules Desharnais, and Béchir Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *LNCS*, pages 100–118. Springer Berlin / Heidelberg, 2010. doi:10.1007/978-3-642-13321-3_8.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104. ACM, 2009. doi:10.1145/1596638.1596650.

Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, February 1992.

Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007. doi:10.1007/978-3-540-76786-2.

Jeremy Gibbons. Unfolding abstract datatypes. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, pages 110–133. Springer-Verlag, 2008. doi:10.1007/978-3-540-70594-9_8.

Erik Hesselink. Generic programming with fixed points for parametrized datatypes. Master's thesis, Universiteit Utrecht, 2009. `https://github.com/hesselink/thesis/`.

Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. doi:10.1016/S0167-6423(02)00025-4.

Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16:451–483, July 2006. doi:10.1017/S0956796806006022.

Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" revolutions. In Tarmo Uustalu, editor, *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 180–208. Springer, 2006. doi:10.1007/11783596_13.

Ralf Hinze and Andres Löh. Generic programming in 3D. *Science of Computer Programming*, 74:590–628, June 2009. doi:10.1016/j.scico.2007.10.006.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. *Electronic Notes in Theoretical Computer Science*, 41(1):5–35, 2001. doi:10.1016/S1571-0661(05)80542-0.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 148–174. Springer-Verlag, 2002. doi:10.1007/3-540-45442-X_10.

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" reloaded. In *Proceedings of the 8th international conference on Functional and Logic Programming*, volume 3945, pages 13–29. Springer-Verlag, 2006. doi:10.1007/11737414_3.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approches to generic programming in Haskell. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *LNCS*, pages 72–149. Springer-Verlag, 2007. doi:10.1007/978-3-540-76786-2_2.

Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez Yakushev. Generic views on data types. In *Proceedings of the 8th International Conference on Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 209–234. Springer, 2006. doi:10.1007/11783596_14.

Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. doi:10.1017/S0956796897002864.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM, 1997. doi:10.1145/263699.263763.

Johan Jeuring, José Pedro Magalhães, and Bastiaan Heeren. Generic programming for domain reasoners. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Proceedings of the 10th Symposium on Trends in Functional Programming*, pages 113–128. Intellect, 2010. ISBN 9781841504056.

Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–308. ACM, 2008. doi:10.1145/1328438.1328475.

Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–12. ACM, 2001. doi:10.1145/378795.378797.

Oleg Kiselyov. Smash your boiler-plate without class and Typeable, 2006. Available at `http://www.haskell.org/pipermail/haskell/2006-August/018353.html`.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003. doi:10.1145/604174.604179.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, pages 244–255. ACM, 2004. doi:10.1145/1016850.1016883.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 204–215. ACM, 2005. doi:10.1145/1086365.1086391.

Andres Löh. *Exploring Generic Haskell*. PhD thesis, Universiteit Utrecht, 2004. `http://igitur-archive.library.uu.nl/dissertations/2004-1130-111344`.

Andres Löh and José Pedro Magalhães. Generic programming with indexed functors. In *Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2011. doi:10.1145/2036918.2036920.

José Pedro Magalhães and W. Bas de Haas. Functional modelling of musical harmony: an experience report. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 156–162. ACM, 2011. doi:10.1145/2034773.2034797.

José Pedro Magalhães and Johan Jeuring. Generic programming for indexed datatypes. In *Proceedings of the 7th ACM SIGPLAN Workshop on Generic Programming*, pages 37–46. ACM, 2011. doi:10.1145/2036918.2036924.

José Pedro Magalhães and Andres Löh. A formal comparison of approaches to datatype-generic programming. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–67. Open Publishing Association, 2012. doi:10.4204/EPTCS.76.6.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the 3rd ACM Haskell Symposium on Haskell*, pages 37–48. ACM, 2010a. doi:10.1145/1863523.1863529.

José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. Optimizing generics is easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 33–42. ACM, 2010b. doi:10.1145/1706356.1706366.

Conor McBride. The derivative of a regular type is its type of one-hole contexts, 2001. Unpublished manuscript, available at `http://www.cs.nott.ac.uk/~ctm/diff.pdf`.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 287–295, 2008. doi:10.1145/1328438.1328474.

Lambert Meertens. Category Theory for Program Construction by Calculation. Unpublished manuscript, September 1995. URL `www.kestrel.edu/home/people/meertens/diverse/ct4pc.pdf`.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer, 1991. doi:10.1007/3540543961_7.

Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 49–60. ACM, 2007. doi:10.1145/1291201.1291208.

Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, The University of Nottingham, November 2007.

Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: Dependent types for relational program derivation. *Journal of Functional Programming*, 19:545–579, September 2009. doi:10.1017/S0956796809007345.

Matthias Neubauer and Peter Thiemann. Type classes with more higher-order polymorphism. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 179–190. ACM, 2002. doi:10.1145/581478.581496.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 13–24. ACM, 2008. doi:10.1145/1411318.1411321.

Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(Special Issue 3–4):375–413, 2010. doi:10.1017/S0956796810000183.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

Ulf Norell and Patrik Jansson. Prototyping generic programming in Template Haskell. In *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer-Verlag, 2004. doi:10.1007/978-3-540-27764-4_17.

Bruno C.d.S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In Henrik Nilsson, editor, *Trends in Functional Programming*, volume 7, pages 199–216. Intellect, 2006.

Will Partain. The `nofib` benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1993.

Simon Peyton Jones, editor. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. doi:10.1017/S0956796803000315. Journal of Functional Programming Special Issue 13(1).

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5):393–433, 2002. doi:10.1017/S0956796802004331.

Simon Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32:3–47, September 1998. doi:10.1016/S0167-6423(97)00029-4.

Simon Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in Haskell. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 267–305. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-642-04652-0_6.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop 2001*, pages 203–233, 2001.

Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*. Springer-Verlag, 2008. doi:10.1007/978-3-540-89330-1_10.

Gabriel dos Reis and Jaakko Järvi. What is generic programming? In *Proceedings of the 1st International Workshop of Library-Centric Software Design*, October 2005.

Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *LNCS*, pages 93–116. Springer, 2010. doi:10.1007/978-3-642-11931-6_5.

Alexey Rodriguez Yakushev, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C.d.S. Oliveira. Comparing libraries for generic programming in Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 111–122. ACM, 2008. doi:10.1145/1411286.1411301.

Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 233–244. ACM, 2009. doi:10.1145/1596550.1596585.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM, 2008. doi:10.1145/1411204.1411215.

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352. ACM, 2009. doi:10.1145/1596550.1596599.

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37:60–75, December 2002. doi:10.1145/636517.636528.

Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002. ISBN 9780201729146.

Wendy Verbruggen, Edsko De Vries, and Arthur Hughes. Formal polytypic programs and proofs. *Journal of Functional Programming*, 20(Special Issue 3-4):213–270, 2010. doi:10.1017/S0956796810000158.

Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989. doi:10.1145/99370.99404.

Malcom Wallace et al. Derived instances—Haskell Prime, April 2007. Available at `http://hackage.haskell.org/trac/haskell-prime/wiki/DerivedInstances`.

Stephanie Weirich. Replib: a library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 1–12. ACM, 2006. doi:10.1145/1159842.1159844.

Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages meets Program Verification*, pages 15–26. ACM, 2010. doi:10.1145/1707790.1707799.

Noel Winstanley and John Meacham. DrIFT user guide, February 2008. Available at `http://repetae.net/computer/haskell/DrIFT/drift.html`.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi:10.1145/2103786.2103795.

Acknowledgements

This thesis is not just the result of four years spent sitting behind a desk at the Department of Information and Computing Sciences at Utrecht University. It is the product of 21 years of academic studies, mixed with all the social interactions that make me who I am, and allowed me to write this thesis. I have several people to thank for this long, enriching journey.

I first met my main supervisor, Johan Jeuring, during an exchange period at Utrecht University during my undergraduate studies, where I had the opportunity to take his Generic Programming course. I asked him about the possibility of doing a PhD at Utrecht University for the first time in November 2004. His positive response led me to contact him again in 2007, and with his research proposal we obtained funding for my PhD. Johan was always available to correct my mistakes, provide me with new insights and research directions, look over my code, or just discuss some random ideas. I could not have asked more from a supervisor, and the success of my PhD is in great part due to him.

I am also extremely thankful, if not even indebted, to Andres Löh for all that I have learnt from him. Andres was a continuous source of good programming practices, challenging problems, good reading material, and is probably responsible for my interest in dependently-typed programming in general, and Agda in particular. I am very glad that he was in Utrecht when I started my PhD, and that our collaboration did not decrease after he moved to Well-Typed.

The lively functional programming research environment in Utrecht is headed by Doaitse Swierstra, who I had the honor of also being supervised by. Doaitse had a profound effect on my understanding of laziness in Haskell, and was always a good source of inspiration and motivation.

After the approval from my supervisors, this thesis was sent to an assessment committee that had the task of deciding "whether the manuscript demonstrates the candidate's capacity for the independent pursuit of research". I am thankful to Jeremy Gibbons,

José Nuno Oliveira, Vincent van Oostrom, Rinus Plasmeijer, and Janis Voigtländer for accepting this task, and for the valuable feedback and comments provided.

I would perhaps not even have followed an academic career in this field if it weren't for the amazing lectures I attended during my undergraduate studies, at Minho University, most notably those by José Nuno Oliveira and Jorge Sousa Pinto. I was lucky to study in a university where Haskell is the first programming language taught, and where its wonders are beautifully exposed by passionate lecturers.

In Utrecht, I had the opportunity to meet lots of people with similar interests that motivated and stimulated me through insightful discussions, and sometimes just helped me relax and unwind. I'm thankful to Arie and Stefan, and Jan and Sean, who shared an office with me during different periods, for making the experience of just sitting behind my desk more enjoyable. I would also like to acknowledge the rest of the Software Technology group, who provided a great research environment to work in: Atze, Jeroen, Jur, and Wishnu. I'm also glad there were plenty of other PhD students around, to play foosball, enjoy the staff "borrels", or, well, discuss research: Alex Elyasov, Alex Gerdes, Alexey, Amir, Thomas, and Jeroen. I'm especially thankful to all those who participated in our weekly reading club sessions, which I always found very valuable.

Chance brought me to once attend a departmental seminar where I heard Bas de Haas talk about his grammar-based approach to musical harmony. Luckily, Bas welcomed my outsider opinion in his field, and was willing to spend a considerable amount of time working on a new approach from the ground up. This gave rise to a very fruitful research collaboration, and allowed me to bring together my love for music and my passion for programming in a way that I did not previously imagine possible. Thank you Bas for all the good moments we've had, and also for those which are still come.

I've also had the chance to collaborate with many authors while working on scientific publications. I am thankful to all my co-authors, to all the anonymous reviewers who provided helpful feedback, and to people who have commented on earlier versions of papers. The amazingly active, friendly, and helpful Haskell and Agda communities also played a big role in motivating me to use these languages.

During the final months of my PhD, I took a short break before starting to write this thesis to work for 12 weeks as a research intern at Microsoft Research Cambridge. I am highly indebted to Simon Peyton Jones for this great opportunity, and for patiently explaining all about the inner details of GHC to me. I have had a great experience with the GHC group at Microsoft Research, also with Dimitrios Vytiniotis and Simon Marlow, and I hope to be able to continue working on improving GHC in the future.

The Portuguese Foundation for Science and Technology (FCT) funded my PhD via the SFRH/BD/35999/2007 grant. I am also thankful to the Department of Information and Computing Sciences at Utrecht University for covering other costs during my PhD, namely regarding participation in international conferences and summer schools, and also for contributing towards the Institute for Programming research and Algorithmics (IPA). The IPA events gave me the opportunity to meet other PhD students in the Netherlands, and I am particularly happy for the good moments at the IPAZoP days.

After my four years in Utrecht I was still left with minor corrections and administrative issues to deal with regarding this thesis. I am thankful to Ralf Hinze, principal

investigator on the project that now funds my research at the University of Oxford, for welcoming me and allowing me some time to finish my thesis. Thanks are also due to the other project team members, Jeremy Gibbons and Nicolas Wu, and to Tom Harper and Daniel James for the friendly and relaxed atmosphere, and valuable native English speaker help.

Many other people contributed to my happiness during my PhD studies; while not directly affecting my research, I am certain of their indirect positive effect on me. I am glad to have been able to sing Bach's *Mass in B minor* with the Utrechts Studenten Koor en Orkest (USKO), and all the amazing chamber choir works with the Utrechtse Studenten Cantorij (in particular Duruflé's *Requiem*). I am especially thankful for all the good moments during our Brazil tour, and to Fokko Oldenhuis for his superb work at conducting and motivating the choir. Arcane thanks are also due to the Ars Magica roleplaying group, with its weekly meetings and pizza sessions; Guandriem thanks Claude, Focus, Fray, Shield, Vleman, and Ysandre. And Minor.

I am lucky to have had one of my earliest childhood friends around in Utrecht. Joana has an uplifting spirit and always reminded me of the warmth of Portugal. Furthermore, she encouraged me to join first the USKO and then the Cantorij, and eventually we found ourselves singing together again, after all those good years back at AMVP. Thank you Joana for all the good moments and for reminding me of the less mechanical and predictable aspects of human life.

Many other friends provided me with support, helped me enjoy life, and gave me the pleasure of sharing their lives with me. Thank you Ernst, Guus, João Guimarães, João Pedro, Pedro Morgado, Pedro Veiga, and Ricardo, for all the moments that we shared, good and bad, days and nights.

It is not easy to put down in words how thankful I am to Derk for everything he has done for me. It is hard to judge if I would have even returned to the Netherlands had I not met him in 2004. Derk single-handedly helped me coping with missing my Portuguese family and friends during my exchange period, and went on to become the most important person in my life. He was always there for me when I doubted my worth, when I did not know how to carry on, and when I needed someone to talk to. Thank you for really knowing and understanding me, for all that you share with me, and for all that you are.

Finally, I thank my family for understanding, supporting, and guiding me through life. The following paragraph is for them, and is therefore written in Portuguese.

Obrigado mãe, pai, irmã, e sobrinha por todo o amor, apoio, e compreensão ao longo da minha vida e das minhas decisões. Obrigado por me darem a oportunidade de aprender música desde tão novo, pelo computador novo a cada dois anos, pelas escolas para onde, a princípio, me enviaram, e, mais tarde, pelas escolas que escolhi, e que aceitaram apoiar. Obrigado por me tornarem no que sou, e pelo orgulho que têm em mim.

Colophon

Author: José Pedro Magalhães
ISBN: 978-90-393-5823-8