Generic Views
INF/SCR-04-67

Generic Views

Stefan Holdermans

Master's thesis
Institute for Information and Computing Sciences
Utrecht University
INF/SCR-04-67

Supervisors:

prof. dr. J. T. Jeuring
dr. A. Löh

Quid videat, nescit: sed quod videt, uritur illo,
atque oculos idem, qui decipit, incitat error.
Credule, quid frustra simulacra fugacia captas?
Quod petis, est nusquam; quod amas, avertere, perdes.
Ista repercussae, quam cernis, imaginis umbra est:
nil habet ista sui; tecum venitque manetque,
tecum discedet—si tu discedere possis!

What he sees, he does not know; but what he sees, he is on fire for,
and the same error, that deceives his eyes, seduces them.
Gullible fool, why do you vainly try to catch a fleeting image?
What you long for is nowhere; what you love you will lose, as soon as
you turn away.
It is the shadow of a reflected image that you perceive:
it has nothing of its own; it has come with you and it stays with you,
it will leave with you—if you are able to leave!

Ovid, *Metamorphoses* III, 430–436.

Abstract

Structural polymorphism allows for generic functions to be defined by induction over the structure of types. In Generic Haskell, the structure of a type is perceived as a nested sum of products. Over the last few years it has been shown that a great amount of generic programs can be defined in terms of this perception. Still, there are applications for which viewing data types as sums of products limits the expressiveness of Generic Haskell. As it turns out, different perceptions of the structure of types enable the definition of generic programs that are more efficient, more elegant, and more expressive. We show how these different perceptions, which we call *generic views*, can be implemented and used in Generic Haskell.

Abstract

Contents

Abstract **vii**

Preface **xv**

1 Introduction **1**
1.1 Type Systems 1
1.2 Related Work 2
1.3 Notes 3

I Preliminaries **5**

2 Setting the Scene **7**
2.1 Syntax 7
2.2 Well-formedness 11
2.3 Semantics 18
2.4 Haskell 98 and Beyond 22
2.5 Notes 24

3 Type-indexed Functions **25**
3.1 Overloading 25
3.2 Dependencies 27
3.3 Notes 30

4 Generics **31**
4.1 Generic Algorithms 31
4.2 Generic Functions 36
4.3 Extensions 40
4.4 Notes 44

II Views **45**

5 Generic Views on Data Types **47**
5.1 Views 47
5.2 Towards Generic Views 50
5.3 Generic Views, Formally 52
5.4 Notes 56

6 Sums and Products **57**
6.1 Standard View 57
6.2 Balanced Encoding 64
6.3 List-like Encoding 69
6.4 Notes 74

7 Recursive Calls in Type Declarations **77**
7.1 Fixed-point View 77
7.2 Recursive View 86
7.3 Notes 89

8 Epilogue **91**
8.1 Other Views 91
8.2 Implementation 94

Bibliography **95**

Index **103**

List of Figures

2.1 Syntax of the expression language 8
2.2 Syntax of the type language 9
2.3 Syntax of the kind language 10
2.4 Order of a kind 11
2.5 Syntax of environments 12
2.6 Syntax of substitutions 12
2.7 Free variables in types 13
2.8 Free variables in expressions 13
2.9 Free variables in type environments 14
2.10 Variables in patterns 14
2.11 Kinding 14
2.12 Typing 15
2.13 Typing patterns 16
2.14 Subsumption 17
2.15 Well-formedness of programs 17
2.16 Well-formedness of type declarations 18
2.17 Syntax of values 18
2.18 Reduction 19
2.19 Pattern matching 20
2.20 Weak-head normal form 21
2.21 Projection functions 21
2.22 Evaluation 21
2.23 Rank of a type 23

5.1 Syntax of the type language, replaces Figure 2.2 53
5.2 Kinding for parameterized types 53

6.1 View types for the standard view 60
6.2 Structural representation of data types in the standard view . . 61
6.3 Structural representation of constructors in the standard view . 61
6.4 Conversions for data types in the standard view 62
6.5 Conversions for constructors in the standard view 63
6.6 View types for the balanced view 66
6.7 Structural representation of data types in the balanced view . . 66
6.8 Structural representation of constructors in the balanced view . 67
6.9 Conversions for data types in the balanced view 68
6.10 Conversions for constructors in the balanced view 68
6.11 View types for the list-like view 72

6.12 Structural representation of data types in the list-like view . . . 73
6.13 Structural representation of constructors in the list-like view . . 73
6.14 Conversions for data types in the list-like view 74
6.15 Conversions for constructors in the list-like view 75

7.1 Pattern functors 83
7.2 View types for the fixed-point view 84
7.3 Structural representation of data types in the fixed-point view . 84
7.4 Conversions for data types in the fixed-point view 85
7.5 Conversions for fields in the fixed-point view 85
7.6 View types for the recursive view 88
7.7 Structural representation of data types in the recursive view . . 88
7.8 Conversions for data types in the recursive view 89
7.9 Conversions for fields in the recursive view 90

List of Tables

2.1 Metavariables 8

5.1 Metavariables, replaces Table 2.1 53

6.1 Generic encoding of days of the week 65
6.2 Manual encoding of days of the week 65

List of Tables

Preface

In my first year as a computer-science student, I took a course on imperative programming. The first serious assignment of the course involved writing a Java program that, given a month and a year, produced a nicely laid-out calendar. I remember struggling with the rules for leap years and the formulas that were used to decide to which day of the week the first day of a specific month corresponds. If I now look at the program I wrote for that assignment, I almost feel embarrassed about the clumsiness of the code. However, back then, that did not matter. What did matter, was that the program produced the correct results—and that I was proud for having it constructed that way.

Later on, of course, it did start to matter what the code I was writing looked like. Moreover, it started to matter how much code I would need to accomplish a given task. I quickly evolved to the stage at which I basically always wanted to do more with less code. Considering this, it is not hard to see why, upon first contact with the language Haskell, I almost immediately fell in love with so-called functional programming. Functional programming languages are remarkably expressive and programs written in a language as Haskell tend to be wonderfully concise.

Still, even in Haskell, reuse of code sometimes is hampered and a single piece of code appears, with slight alterations, in multiple parts of a program. This is one of the issues addressed by the research on generic programming. I learned about this field through a series of lectures by Johan Jeuring. From then on, it was crystal clear to me that I wanted to center my thesis project around generic programming.

And now, a few years later, I'm more than happy to have indeed finished a thesis project that focussed on generic-programming techniques.

Organization

The thesis is organized as follows:

The first chapter provides a brief introduction to functional generic programming and gives an overview of related work in the field.

The next three chapters lay the groundwork for the development in later chapters. In Chapter 2 a formal language is presented, that is used throughout the thesis in formal definitions and listings of algorithms. Chapter 3 describes how Generic Haskell extends Haskell with the notion of type-indexed functions. In Chapter 4 it is shown how type-indexed functions are used in generic programming.

In the second part of the thesis—consisting of Chapters 5, 6, 7, and 8—the spotlight is on our main topic: views on data types. After the concept of a

generic view is introduced in Chapter 5, Chapters 6 and 7 provide worked-out examples of several archetypical views. Finally, Chapter 8 concludes by considering some other possible views and discussing the implementation of a language extension that supports the simultaneous use of multiple views on data types.

Acknowledgments

A more concice presentation of the material covered in this thesis is provided in the forthcoming paper *Generic Views on Data Types* [42].

Lots of people contributed in some way to the writing of this thesis: I am very grateful to Johan Jeuring and Andres Löh for their supervision. Without their help and suggestions I would probably still be trying all kinds of fruitless alternative solutions to problems that could have been avoided altogether by just being a little more creative and a little less narrow-minded. They made this thesis project the rich and insightful experience it was. Thanks are due to Marc de Hoon for helping me with the translation of the motto. I would also like to thank my former collegues at Unit 4 Agresso for showing me how software development is done in practice; Theo Albers and Rein van Erkel, in particular, have shown me that craftsmanship and search for innovation make an excellent combination. Over the years, Joost Ronkes Agerbeek has been a tremendous source of inspiration; I hope we will continue having long and passionate conversations about writing software and all that is related. Finally, my deepest feelings of gratitude go to my friends and family for their unremitting support—and to Kim, for believing in me.

Dordrecht, April 2005 Stefan Holdermans

Chapter 1

Introduction

This introductory chapter briefly introduces the idea behind functional generic programming and mentions some related work.

1.1 Type Systems

Types play an important rôle in modern programming. A type system provides a way to syntactically prove the absence of certain programming errors. Furthermore, types may help structuring complex programs and provide machine-checkable documentation.

The material in this thesis centers around the functional programming language Haskell [75]. Haskell makes use of a Hindley-Milner type system [31, 65, 20]—or, more precise, of an extension thereof. In such a system, it is not necessary to explicitly write type signatures for each function. Instead, types are automatically inferred by the compiler. The demand that all types can be inferred, however, puts some restrictions on the programs that can be written in a language. Therefore, most implementations of the Haskell language provide extensions that break some of these restrictions, but require explicit type signatures to be supplied by the programmer.

Still, even with these extensions, Haskell's type system sometimes gets in the way. It is for example not possible to write a function that tests for equality that works for all types. This is were generic programming steps in.

Generic Programming

Generic Haskell is a superset of Haskell that supports the definition of so-called *generic functions*. These are functions that are defined by induction on the structure of types. In contrast, ordinary, non-generic functions are usually defined by induction on the structure of values.

As a result, in Generic Haskell it is possible to define an equality function that works for (almost) all types.

Other approaches to generic programming also make use of the structure of types. These approaches differ in the way how this structure is perceived. Different approaches use different perceptions and, hence, give rise to different

styles of generic programming. Each style has its advantages and disadvantages. Some perceptions of the structure of types allow the definition of generic functions that are impossible or at least hard to define in other approaches. Other perceptions allow the definition of more efficient generic functions.

In this thesis, we focus on bringing together different perceptions of type structure into a single framework for generic programming. To this end, we study how the structure of types is perceived in Generic Haskell and we propose to extend the language in such a way that it allows for multiple perceptions to coexist.

1.2 Related Work

The following subsections describe some related work in the field of generic programming.

Type Classes

Type classes [90] in Haskell essentially provide support for *overloading*: they allow a single name to be used for two or more different functions. For some special overloaded functions, such as equality checks, Haskell is capable of automatically deriving the appropriate function definition. The set of functions for which this feature is available is, however, limited.

Hinze and Peyton Jones [41] propose to remove this limitation by allowing type-class functions to be defined by induction on the structure of types.

Intensional Type Analysis

Intensional type analysis [29, 91] provides a **typecase** construct that supports case analyses on type arguments. The main difference with Generic Haskell lies in the fact that intensional type analysis performs the case analyses at the run time of a program, whereas Generic Haskell does it at compile time. On the other hand, intensional type analysis supports higher-order generic functions, which cannot be defined in Generic Haskell.

Pattern Calculus

Jay's *pattern calculus* [50], which is based on the *constructor calculus* [49], extends the concept of case analysis on value arguments by permitting patterns of multiple types to be used with a single **case** construct. Furthermore, some special patterns are provided, such as constructor application, that can be used to match against values of any type.

Dependent Types

It has been shown that Generic Haskell can be simulated in Haskell-based languages that support programming with *dependent types*. An example of such a language is Cayenne [8].

Altenkirch and McBride [4] provide some examples of generic functions that can be encoded in a dependently typed language; some of the language

features they use, however, are not available in Cayenne. Consequently, this particular style of programming with dependent types set the scene for the development of a new programming language, named Epigram [62].

1.3 Notes

There are a great number of tutorial texts available on the Haskell language, most notably Bird's [11] and Hudak's [43].

Excellent introductory material on types and type systems is provided in the textbooks of Mitchell [66] and Pierce [77]. A successor of the latter text [78] provides an in-depth overview of some of the more advanced issues regarding Hindley-Milner type systems [79].

A thorough introduction to the theory behind functional generic programming is given by Backhouse et al. [9].

The term *generic programming* is a bit overloaded. In this thesis, it is used to refer to the style of programming in which functions are defined by induction over the structure of types. However, in the world of object-oriented programming, the term usually refers to language features that support parametric polymorphism.

Part I

Preliminaries

Chapter 2

Setting the Scene

Throughout this thesis, generic-programming concepts are studied in the context of Generic Haskell, an extension of the functional programming language Haskell. However, Haskell itself is too rich a language to be suitable for the formal analyses we want to perform in later chapters. Therefore, in this chapter, a small core language is presented that serves as the object language in the formal parts of this thesis. This language is only a slight variation on the core language used in Löh's thesis [58].

An important aspect of the core language is that, although small, it is still rich enough to illustrate our approach to generic programming. More specific, the language is as expressive as Haskell: advanced constructs in Haskell can be regarded as mere syntactic sugar on top of constructions in the core language. This can be made concrete by showing a translation from Haskell programs to core-language programs. However, such a translation is left implicit here.

The chapter is organized as follows. Section 2.1 provides a context-free grammar for the core language; because types play an important rôle in generic programming, we mainly focus on the syntax of types and kinds. In Section 2.2 we show how programs are typed. Operational semantics are provided in Section 2.3. Section 2.4 discusses Haskell, the language used in the informal parts of the thesis.

2.1 Syntax

In this section we introduce the syntax of the core language. The syntax is split into an expression language, a type language, and a kind language. Table 2.1 provides an overview of the used metavariables. It is assumed that we have an infinite amount of symbols available that can be used as variables and constructors.

Programs and Expressions

Figure 2.1 shows the syntax of programs in the core language. A program consists of zero or more type declarations and a single expression: the main function. Type declarations are dealt with in the next subsection; here we focus on expressions.

a	type variable	v	value
d	value declaration	x	variable
e	expression	C	data constructor
i	natural number	D	type declaration
j	natural number	P	program
k	natural number	T	type constructor
ℓ	natural number	κ	kind
m	natural number	φ	value substitution
n	natural number	ψ	type substitution
p	pattern	Γ	type environment
t	type	K	kind environment

Table 2.1: Metavariables

Programs

P	$::=$	$\{D_i;\}^{i\in 1..n}\ e$	type declarations and main expression

Value declarations

d	$::=$	$x = e$	function declaration

Expressions

e	$::=$	x	variable
	$\mid$	C	data constructor
	$\mid$	$\lambda x\ .\ e$	abstraction
	$\mid$	$(e_1\ e_2)$	application
	$\mid$	$\mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i\in 1..n}$	case
	$\mid$	$(\mathbf{fix}\ e)$	fixed point
	$\mid$	$\mathbf{let}\ \{d_i\}_{;}^{i\in 1..n}\ \mathbf{in}\ e$	let
	$\mid$	$\bot$	bottom

Patterns

p	$::=$	x	variable pattern
	$\mid$	$(C\ \{p_i\}^{i\in 1..n})$	constructor pattern

Figure 2.1: Syntax of the expression language

Type declarations

$$D \quad ::= \quad \textbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}^{j \in 1..m}_{|}$$

algebraic data type

Types

t	$::=$	a	type variable
	$\mid$	T	type constructor
	$\mid$	$(t_1\ t_2)$	type application
	$\mid$	$\forall a :: \kappa\ .\ t$	universal quantification

Figure 2.2: Syntax of the type language

The basic building blocks for expressions are variables and constructors. Functions are introduced and eliminated by means of, respectively, lambda abstractions and applications. To minimize the number of parentheses needed, we adopt the usual conventions for writing abstractions and applications, i.e., the scope of a lambda extends as far to the right as possible and application associates to the left.

Pattern matching is done through **case** expressions. A **case** expression consists of a head expression, i.e., the expression to be analyzed, and zero or more arms, each containing a pattern and an expression. A pattern is either a variable or an applied constructor.

A fixed-point operator is provided, but it is only included to accomodate the **let** construct, which binds a group of mutually recursive function definitions.

The expression language is completed with a diverging operator, $\perp$.

Types and Kinds

The syntax of the type language is shown in Figure 2.2. Types are built from type variables and type constructors. Futhermore, a type can be applied to another type and we have universal quantification over type variables. (Value-level type applications are implicit.)

New types are introduced by means of **data** declarations. Such a declaration associates a, possibly parameterized, type constructor with zero or more data constructors, each of which has zero or more fields.

For instance, the following declaration defines the type of Boolean values:

data Bool = *False* | *True*.

The type constructor Bool has no parameters and two data constructors, both of which have no fields. In general, parameterless type constructors with nothing but nullary data constructors are called *enumerations*.

Record types have only one data constructor; for example,

data Fork = Λa :: $\star$. *Fork* a a.

Here, Fork is a unary type constructor that has one binary data constructor, *Fork*. It is perfectly all right to simultaneously assign the same name to a type

Kinds			
κ	$::=$	$\star$	kind of proper types
	$\mid$	$\kappa_1 \to \kappa_2$	function kind

Figure 2.3: Syntax of the kind language

constructor and a data constructor, since type constructors and data constructors constitute different namespaces.

Data constructors are used to construct expressions. For instance,

$$\mathit{Fork}\ \mathit{False}\ \mathit{True}$$

is an expression of type Fork Bool. The expression

$$\mathit{Fork}\ (\mathit{Fork}\ \mathit{True}\ \mathit{True})\ (\mathit{Fork}\ \mathit{True}\ \mathit{False})$$

has type Fork (Fork Bool).

The formal parameter, a, in the declaration of the type constructor Fork is decorated with a kind annotation. A *kind* can be thought of as the type of a type: kinds impose structure on the level of types, just like types impose structure on the level of expressions. The kind language of the core language is presented in Figure 2.3. Kinds are built from two kind constructors: the nullary constructor $\star$ and the binary constructor $(\to)$. Kind $\star$ is reserved for types that can be assigned to expressions. Types with kind $\kappa_1 \to \kappa_2$ take types of kind κ_1 to types of kind κ_2. For instance, the type Bool has kind $\star$, while Fork has kind $\star \to \star$. In the type language, kind annotations appear in **data** declarations and in universal quantifications.

Besides finite types like Bool and Fork, type declarations can also be used to define *recursive types*. The most frequently employed example of a recursive data type is the type of parametric lists:

$$\textbf{data}\ \mathrm{List} = \Lambda \mathrm{a} :: \star\ .\ \mathit{Nil} \mid \mathit{Cons}\ \mathrm{a}\ (\mathrm{List}\ \mathrm{a}).$$

The type constructor List has kind $\star \to \star$.

The type List is used in the declaration of the type Rose of rose trees [11],

$$\textbf{data}\ \mathrm{Rose} = \Lambda \mathrm{a} :: \star\ .\ \mathit{Branch}\ \mathrm{a}\ (\mathrm{List}\ (\mathrm{Rose}\ \mathrm{a})).$$

Abstracting over List, we obtain a generalization of Rose,

$$\textbf{data}\ \mathrm{GRose} = \Lambda \mathrm{f} :: \star \to \star\ .\ \Lambda \mathrm{a} :: \star\ .\ \mathit{GBranch}\ \mathrm{a}\ (\mathrm{f}\ (\mathrm{GRose}\ \mathrm{f}\ \mathrm{a})),$$

that takes two parameters, of which the first, f, ranges over type constructors of kind $\star \to \star$. The kind of GRose, $(\star \to \star) \to \star \to \star$, is an example of a second-order kind. Figure 2.4 shows how to determine the order of a kind.

The kind of GRose illustrates that the kind constructor $(\to)$ associates to the right, i.e., binary type constructors are curried. Another example comes from the type Tree of external binary search trees,

$$\textbf{data}\ \mathrm{Tree} = \Lambda \mathrm{a} :: \star\ .\ \Lambda \mathrm{b} :: \star\ .\ \mathit{Tip}\ \mathrm{a} \mid \mathit{Node}\ (\mathrm{Tree}\ \mathrm{a}\ \mathrm{b})\ \mathrm{b}\ (\mathrm{Tree}\ \mathrm{a}\ \mathrm{b}),$$

$$\boxed{\mathsf{order}(\kappa) \equiv n}$$

$$\frac{}{\mathsf{order}(\star) \equiv 0} \quad \text{(ord-star)} \qquad \frac{\mathsf{order}(\kappa_1) \equiv m \qquad \mathsf{order}(\kappa_2) \equiv n}{\mathsf{order}(\kappa_1 \to \kappa_2) \equiv \max\{1+m, n\}} \quad \text{(ord-fun)}$$

Figure 2.4: Order of a kind

which has kind $\star \to \star \to \star$.

The data types we have seen so far are all examples of regular or uniform types. A type is regular if its definition does not contain any recursive calls that involve a change in type parameters. An example of a non-regular or nested data type [12] is the type of perfectly balanced, binary leaf trees [34],

$$\textbf{data}\ \text{Perfect} = \Lambda \text{a} :: \star\ .\ \mathit{ZeroP}\ \text{a} \mid \mathit{SuccP}\ (\text{Perfect}\ (\text{Fork a})).$$

A type that cannot be defined by means of a **data** declaration is the function-space constructor $(\to)$, not to be mistaken with the kind constructor of the same name. We therefore assume that $(\to)$ is a predefined type of kind $\star \to \star \to \star$. Instead of $((\to)\ t_1\ t_2)$, we write $t_1 \to t_2$. Furthermore, for reasons that become clear in Section 2.3, we assume that we have an infinite number of tuple types at our disposal, behaving as if they were defined by instances of the following scheme:

$$\textbf{data}\ (\{,\}^{i \in 1..n-1}) = \{\Lambda a_i\ .\ \}^{i \in 1..n}\ (\{,\}^{i \in 1..n-1})\ \{a_i\}^{i \in 1..n},$$

where $n \in 2..$ For instance, the declaration for ordered pairs or 2-tuples is given by

$$\textbf{data}\ (,) = \Lambda \text{a} :: \star\ .\ \Lambda \text{b} :: \star\ .\ (,)\ \text{a b}.$$

We write (t_1, t_2) for $(,)\ t_1\ t_2$ and (e_1, e_2) for $(,)\ e_1\ e_2$. Similar conventions are adopted for tuple types of higher arity.

2.2 Well-formedness

The core language is actually significantly smaller than the set of sentences that are described by the grammars in the previous section. In the sequel we only consider well-formed programs; in this section we define which programs are well-formed. In particular, a program is well-formed if its type declarations are well-formed and its main expression is well-typed.

In addition to the rules that are presented in this section, it is assumed that a well-formed program does not contain any conflicting declarations and that any two variables that appear in the same pattern are distinct.

Environments and Substitutions

For defining the well-formedness of programs, we make use of two types of environments. Their syntaxes are shown in Figure 2.5. First, we have kind

Kind environments

K	::=	ε	empty kind environment
	|	$\mathrm{K}, a :: \kappa$	type-variable binding
	|	$\mathrm{K}, T :: \kappa$	type-constructor binding

Type environments

Γ	::=	ε	empty type environment
	|	$\Gamma, x :: t$	variable binding
	|	$\Gamma, C :: t$	data-constructor binding

Figure 2.5: Syntax of environments

Type substitutions

ψ	::=	ε	empty type substitution
	|	$[t \,/\, a] \circ \psi$	type-variable substitution

Value substitutions

φ	::=	ε	empty value substitution
	|	$[e \,/\, x] \circ \varphi$	variable substitution
	|	$\bot$	bottom substitution

Figure 2.6: Syntax of substitutions

environments K, containing bindings of the forms $a :: \kappa$ and $T :: \kappa$, for associating kinds with type variables and type constructors. Second, we have type environments Γ, containing bindings of the forms $x :: t$ and $C :: t$, for associating types with variables and data constructors. Empty environments are denoted by ε.

Furthermore, we need a notion of substitution. Figure 2.6 shows the syntax of type substitutions ψ and value substitutions φ. The substitution that replaces the free occurences of a type variable a by a type t is denoted $[t \,/\, a]$; similarly, the substitution that replaces the free occurrences of a variable x by an expression e is denoted $[e \,/\, x]$. We write $\psi_2 \circ \psi_1$ for the composition of two type substitutions ψ_1 and ψ_2, and $\varphi_2 \circ \varphi_1$ for the composition of two value substitutions φ_1 and φ_2; empty substitutions occur as ε. Additionally, we distinguish a special value substitution, $\bot$, that maps all free variables in a expression to $\bot$. Application of type substitutions, written $(\psi\ t)$, and value substitutions, written $(\varphi\ e)$, is not formally defined here, but is assumed to preserve free variables, i.e., to perform alpha conversion when needed.

The free variables in types and expressions are given by the metafunctions ftv and fev in Figures 2.7 and 2.8. Figure 2.9 shows how the collection of free type variables is extended to environments. The variables that occur in patterns are collected by the function pv in Figure 2.10.

Kinding

Types are checked to be well-kinded according to the rules in Figure 2.11. These rules take the form

$$\boxed{\mathsf{ftv}(t) \equiv \{a_i\}_{,}^{i \in 1..n}}$$

$$\frac{}{\mathsf{ftv}(a) \equiv a} \text{ (ftv-var)} \qquad \frac{}{\mathsf{ftv}(T) \equiv \varepsilon} \text{ (ftv-con)}$$

$$\frac{}{\mathsf{ftv}(t_1\ t_2) \equiv \mathsf{ftv}(t_1), \mathsf{ftv}(t_2)} \text{ (ftv-app)} \qquad \frac{}{\mathsf{ftv}(\forall a :: \kappa\ .\ t) \equiv \mathsf{ftv}(t) \setminus a} \text{ (ftv-forall)}$$

Figure 2.7: Free variables in types

$$\boxed{\mathsf{fev}(e) \equiv \{x_i\}_{,}^{i \in 1..n}}$$

$$\frac{}{\mathsf{fev}(x) \equiv x} \text{ (fev-var)} \qquad \frac{}{\mathsf{fev}(C) \equiv \varepsilon} \text{ (fev-con)}$$

$$\frac{}{\mathsf{fev}(\lambda x\ .\ e) \equiv \mathsf{fev}(e) \setminus x} \text{ (fev-abs)}$$

$$\frac{}{\mathsf{fev}(e_1\ e_2) \equiv \mathsf{fev}(e_1), \mathsf{fev}(e_2)} \text{ (fev-app)}$$

$$\frac{}{\mathsf{fev}(\mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i \in 1..n}) \equiv \mathsf{fev}(e_0)\ \{, \mathsf{fev}(e_i) \setminus \mathsf{pv}(p_i)\}^{i \in 1..n}} \text{ (fev-case)}$$

$$\frac{}{\mathsf{fev}(\mathbf{fix}\ e) \equiv \mathsf{fev}(e)} \text{ (fev-fix)}$$

$$\frac{}{\mathsf{fev}(\mathbf{let}\ (\{x_i = e_i\}_{;}^{i \in 1..n}\ \mathbf{in}\ e_0)) \equiv \mathsf{fev}(e_0)\ \{, \mathsf{fev}(e_i) \setminus x_i\}^{i \in 1..n}} \text{ (fev-let)}$$

$$\frac{}{\mathsf{fev}(\bot) \equiv \varepsilon} \text{ (fev-bot)}$$

Figure 2.8: Free variables in expressions

$$\boxed{\mathsf{ftv}(\Gamma) \equiv \{a_i\}_{,}^{i \in 1..n}}$$

$$\frac{}{\mathsf{ftv}(\varepsilon) \equiv \varepsilon} \quad \text{(ftvenv-empty)} \qquad \frac{}{\mathsf{ftv}(\Gamma, x :: t) \equiv \mathsf{ftv}(\Gamma), \mathsf{ftv}(t)} \quad \text{(ftvenv-var)}$$

$$\frac{}{\mathsf{ftv}(\Gamma, C :: t) \equiv \mathsf{ftv}(\Gamma), \mathsf{ftv}(t)} \quad \text{(ftvenv-con)}$$

Figure 2.9: Free variables in type environments

$$\boxed{\mathsf{pv}(p) \equiv \{x_i\}_{,}^{i \in 1..n}}$$

$$\frac{}{\mathsf{pv}(x) \equiv x} \quad \text{(pv-var)} \qquad \frac{}{\mathsf{pv}(C\ \{p_i\}^{i \in 1..n}) \equiv \{\mathsf{pv}(p_i)\}_{,}^{i \in 1..n}} \quad \text{(pv-con)}$$

Figure 2.10: Variables in patterns

$$\boxed{\mathrm{K} \vdash t :: \kappa}$$

$$\frac{a :: \kappa \in \mathrm{K}}{\mathrm{K} \vdash a :: \kappa} \quad \text{(k-var)} \qquad \frac{T :: \kappa \in \mathrm{K}}{\mathrm{K} \vdash T :: \kappa} \quad \text{(k-con)}$$

$$\frac{\mathrm{K} \vdash t_1 :: \kappa_1 \to \kappa_2 \qquad \mathrm{K} \vdash t_2 :: \kappa_1}{\mathrm{K} \vdash (t_1\ t_2) :: \kappa_2} \quad \text{(k-app)} \qquad \frac{\mathrm{K}, a :: \kappa \vdash t :: \star}{\mathrm{K} \vdash \forall a :: \kappa\ .\ t :: \star} \quad \text{(k-forall)}$$

Figure 2.11: Kinding

$$\boxed{\mathrm{K};\Gamma \vdash e :: t}$$

$$\frac{x :: t \in \Gamma}{\mathrm{K};\Gamma \vdash x :: t}\ \text{(t-var)} \qquad \frac{C :: t \in \Gamma}{\mathrm{K};\Gamma \vdash C :: t}\ \text{(t-con)}$$

$$\frac{\mathrm{K} \vdash t_1 :: \star \qquad \mathrm{K};\Gamma, x :: t_1 \vdash e :: t_2}{\mathrm{K};\Gamma \vdash \lambda x\,.\,e :: t_1 \to t_2}\ \text{(t-abs)} \qquad \frac{\mathrm{K};\Gamma \vdash e_1 :: t_1 \to t_2 \qquad \mathrm{K};\Gamma \vdash e_2 :: t_1}{\mathrm{K};\Gamma \vdash (e_1\ e_2) :: t_2}\ \text{(t-app)}$$

$$\frac{\mathrm{K};\Gamma \vdash e_0 :: t_0 \qquad \{\Gamma \vdash^{pat} p_i :: t_0 \rightsquigarrow \Gamma_i\}^{i \in 1..n} \qquad \{\mathrm{K};\Gamma, \Gamma_i \vdash e_i :: t\}^{i \in 1..n}}{\mathrm{K};\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_{;}^{i \in 1..n} :: t}\ \text{(t-case)}$$

$$\frac{\mathrm{K};\Gamma \vdash e :: t \to t}{\mathrm{K};\Gamma \vdash \mathbf{fix}\ e :: t}\ \text{(t-fix)} \qquad \frac{\{\mathrm{K} \vdash t_i :: \star\}^{i \in 1..n} \qquad \Gamma' \equiv \Gamma\ \{, x_i :: t_i\}^{i \in 1..n} \qquad \{\mathrm{K};\Gamma' \vdash e_i :: t_i\}^{i \in 0..n}}{\mathrm{K};\Gamma \vdash \mathbf{let}\ \{x_i = e_i\}_{;}^{i \in 1..n}\ \mathbf{in}\ e_0 :: t_0}\ \text{(t-let)}$$

$$\frac{\mathrm{K} \vdash t :: \star}{\mathrm{K};\Gamma \vdash \bot :: t}\ \text{(t-bot)}$$

$$\frac{a \notin \mathsf{ftv}(\Gamma) \qquad \mathrm{K}, a :: \kappa;\Gamma \vdash e :: t}{\mathrm{K};\Gamma \vdash e :: \forall a :: \kappa\,.\,t}\ \text{(t-gen)} \qquad \frac{\mathrm{K};\Gamma \vdash e :: t_1 \qquad \mathrm{K} \vdash t_1 \leqslant t_2}{\mathrm{K};\Gamma \vdash e :: t_2}\ \text{(t-subs)}$$

Figure 2.12: Typing

$$\mathrm{K} \vdash t :: \kappa,$$

expressing that, under kind environment K, type t is assigned kind κ.

The kind of a type variable follows from looking it up in the kind environment. The same holds for type constructors. For type applications $(t_1\ t_2)$, we require t_1 to have a function kind $\kappa_1 \to \kappa_2$ and t_2 to have a kind that matches κ_1. The application then has kind κ_2. For universal quantifications $\forall a :: \kappa\,.\,t$, the kind environent is extended with the binding $a :: \kappa$ to obtain a kind for t. The resulting kind, which is required to be $\star$, is assigned to the quantification.

Typing

The typing rules for the core language, which are of the form

$$\mathrm{K}, \Gamma \vdash e :: t,$$

$$\boxed{\Gamma_1 \vdash^{pat} p :: t \rightsquigarrow \Gamma_2}$$

$$\frac{}{\Gamma \vdash^{pat} x :: t \rightsquigarrow x :: t} \quad \text{(p-var)}$$

$$\frac{C :: \{t_i \rightarrow\}^{i \in 1..n} \; t_0 \in \Gamma \qquad \{\Gamma \vdash^{pat} p_i :: t_i \rightsquigarrow \Gamma_i\}^{i \in 1..n}}{\Gamma \vdash^{pat} (C \; \{p_i\}^{i \in 1..n}) :: t_0 \rightsquigarrow \{\Gamma_i\}_{,}^{i \in 1..n}} \quad \text{(p-con)}$$

Figure 2.13: Typing patterns

saying that the expression e has type t under kind environment K and type environment Γ, are shown in Figure 2.12.

The types of variables and constructors are looked up in the type environment.

Since variables act as placeholders for expressions, lambda abstraction is performed over variables of $\star$-kinded types exclusively. An abstraction $\lambda x \,.\, e$ has function type $t_1 \rightarrow t_2$ if, assuming type t_1 for the variable x, the expression e has type t_2.

The rule for applications mirrors the rule for kinding type applications in the preceding subsection.

For **case** constructs we start by obtaining a type t_0 for the head of the expression, e_0. Then the types of the patterns are matched against t_0, using a subsidiary judgement of the form

$$\Gamma_1 \vdash^{pat} p :: t \rightsquigarrow \Gamma_2,$$

expressing that matching the type of a pattern p agaist a type t, under environment Γ_1, results in an environment Γ_2 that contains bindings for the variables in p. The rules for judgements of this kind are depicted in Figure 2.13. The derived bindings are used to type the expressions that appear in the arms of the **case** construct. If these expressions share the same type, this type is assigned to the whole **case** expression.

The typing rule for the fixed-point operator is straightforward. If e is a function of type $t \rightarrow t$, i.e., the domain and range of e coincide, then $(\textbf{fix}\; e)$ has type t.

For **let** expressions, we construct an extended type environment with bindings for the declared variables, and use it to type the bound expressions and the body. The type of the body is then used as the type of the whole expression. Of course, the types of the declared variables should all have kind $\star$.

The diverging operator $\perp$ can have any $\star$-kinded type.

The type of an expression can be generalized by quantifying over a type variable, if the variable does not occur in the type environment, i.e., type variables that do occur in the type environment are considered monomorphic.

If a type t_1 can be derived for an expression e, then we can consequently derive any type t_2 by which t_1 is subsumed for e. The subsumption relation is shown in Figure 2.14, with judgements of the form

$$\boxed{\mathrm{K} \vdash t_1 \leqslant t_2}$$

$$\frac{}{\mathrm{K} \vdash t \leqslant t} \quad \text{(s-refl)} \qquad \frac{\mathrm{K} \vdash t_3 \leqslant t_1 \qquad \mathrm{K} \vdash t_2 \leqslant t_4}{\mathrm{K} \vdash t_1 \to t_2 \leqslant t_3 \to t_4} \quad \text{(s-fun)}$$

$$\frac{a \notin \mathsf{ftv}(t_1) \qquad \mathrm{K}, a :: \kappa \vdash t_1 \leqslant t_2}{\mathrm{K} \vdash t_1 \leqslant \forall a :: \kappa \,.\, t_2} \quad \text{(s-skol)} \qquad \frac{\mathrm{K} \vdash t_0 :: \kappa \qquad \mathrm{K} \vdash [t_0 \,/\, a]\, t_1 \leqslant t_2}{\mathrm{K} \vdash \forall a :: \kappa \,.\, t_1 \leqslant t_2} \quad \text{(s-inst)}$$

Figure 2.14: Subsumption

$$\boxed{\mathrm{K}; \Gamma \vdash P :: t}$$

$$\frac{\{\mathrm{K} \vdash D_i \rightsquigarrow \mathrm{K}_i; \Gamma_i\}^{i \in 1..n} \qquad \mathrm{K} \equiv \mathrm{K}_0\ \{, \mathrm{K}_i\}^{i \in 1..n} \qquad \Gamma \equiv \Gamma_0\ \{, \Gamma_i\}^{i \in 1..n} \qquad \mathrm{K}; \Gamma \vdash e :: t}{\mathrm{K}_0; \Gamma_0 \vdash \{D_i;\}^{i \in 1..n}\ e :: t} \quad \text{(wf-prog)}$$

Figure 2.15: Well-formedness of programs

$$\mathrm{K} \vdash t_1 \leqslant t_2,$$

indicating that a type t_1 is subsumed by a type t_2 under kind environment K.

Well-formedness of Programs

Using the rules for kinding and typing from the previous subsection, we can define well-formedness for programs, as depicted in Figure 2.15. The judgement that a program is well-formed takes the form

$$\mathrm{K}; \Gamma \vdash P :: t,$$

which expresses that, under initial environments K and Γ, a type t can be assigned to the main expression of a program P.

The initial environments can contain external, predefined types and functions. The initial kind environment should at least contain appropriate bindings for the function-space constructor and the infinite amount of tuple types. The initial type environment is required to contain the bindings for the data constructors of the tuple types. Furthermore, the initial type environment should only bind variables and constructors to types that, under the initial kind environment, have kind $\star$.

The rule for well-formedness of programs in Figure 2.15 makes use of a subsidiary rule for the well-formedness of type declarations. The latter, shown in Figure 2.16 is of the form

$$\boxed{\mathrm{K}_1 \vdash D \rightsquigarrow \mathrm{K}_2; \Gamma}$$

$$\frac{\begin{array}{c} D \equiv \mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m} \\ \{\{\mathrm{K}\ \{, a_i :: \kappa_i\}^{i \in 1..\ell} \vdash t_{j,k} :: \star\}^{k \in 1..n_j}\}^{j \in 1..m} \\ \Gamma \equiv \{C_j :: \{\forall a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ \{t_{j,k} \rightarrow\}^{k \in 1..n_j}\ T\ \{a_i\}^{i \in 1..\ell}\}_{,}^{j \in 1..m} \end{array}}{\mathrm{K} \vdash D \rightsquigarrow T :: \{\kappa_i \rightarrow\}^{i \in 1..\ell} \star; \Gamma} \quad \text{(wf-data)}$$

Figure 2.16: Well-formedness of type declarations

Values			
v	$::=$	$(C\ \{v_i\}^{i \in 1..n})$	constructor value
	$\mid$	$\lambda x\ .\ e$	function value
	$\mid$	$\bot$	run-time failure

Figure 2.17: Syntax of values

$$\mathrm{K}_1 \vdash D \rightsquigarrow \mathrm{K}_2; \Gamma,$$

meaning that, under kind environment K_1, a well-formed type declaration D gives rise to a kind environment K_2 and a type environment Γ that contain bindings for, respectively, the type constructor and the data constructors declared by D.

Note that the well-formedness of the type declarations in a program is determined under an extended kind environment that contains, besides the initial kind environment, all kind environments that arise from the well-formedness of the declarations, i.e., the type declarations may be mutually recursive. The extended kind environment, as well as the type environment that is produced by combining the initial type environment with the bindings for data constructors, is used for typing the main expression.

2.3 Semantics

In this section we define the semantics of a well-formed program by means of a reduction system.

To this end, we identify a subset of expressions that contains possible program results. The members of this subset are called *values*; their syntax is given in Figure 2.17. A value can be a constructor applied to zero or more subvalues, an unapplied lambda abstraction, or $\bot$, indicating run-time failure.

Reduction

In Figure 2.18, we present a set of rules for the one-step reduction of expressions. Each rule has the form

$$\boxed{\vdash e_1 \rightarrowtail e_2}$$

$$\frac{n \in 1\,..\qquad e_1 \rightarrowtail e_1'}{\vdash (C\ \{v_i\}^{i\in 1..m}\ \{e_i\}^{i\in 1..n}) \rightarrowtail (C\ \{v_i\}^{i\in 1..m}\ e_1'\ \{e_i\}^{i\in 2..n})} \quad \text{(r-con)}$$

$$\frac{}{\vdash ((\lambda x\,.\,e_1)\ e_2) \rightarrowtail [e_2\ /\ x]\ e_1} \quad \text{(r-app-1)}$$

$$\frac{}{\vdash (\bot\ e) \rightarrowtail \bot} \quad \text{(r-app-2)} \qquad \frac{\vdash e_1 \rightarrowtail e_1'}{(e_1\ e_2) \rightarrowtail (e_1'\ e_2)} \quad \text{(r-app-3)}$$

$$\frac{}{\vdash \textbf{case}\ e_0\ \textbf{of}\ \varepsilon \rightarrowtail \bot} \quad \text{(r-case-1)}$$

$$\frac{n \in 1\,..\qquad \vdash^{match} p_1 \leftarrow e \rightsquigarrow \bot}{\vdash \textbf{case}\ e_0\ \textbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 1..n} \rightarrowtail \textbf{case}\ e_0\ \textbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 2..n}} \quad \text{(r-case-2)}$$

$$\frac{n \in 1\,..\qquad \vdash^{match} p_1 \leftarrow e \rightsquigarrow \varphi \qquad \varphi \not\equiv \bot}{\vdash \textbf{case}\ e_0\ \textbf{of}\ \{p_i \rightarrow e_i\}_{;}^{i\in 1..n} \rightarrowtail (\varphi\ e_1)} \quad \text{(r-case-3)}$$

$$\frac{}{\vdash \textbf{fix}\ e \rightarrowtail (e\ (\textbf{fix}\ e))} \quad \text{(r-fix)}$$

$$\frac{x_0 \notin \{\mathsf{fev}(e_i),\}^{i\in 1..n}\ \mathsf{fev}(e_0)}{\begin{array}{l}\vdash \textbf{let}\ \{x_i = e_i\}_{;}^{i\in 1..n}\ \textbf{in}\ e_0 \rightarrowtail \\ \quad \textbf{case}\ (\textbf{fix}\ \lambda x_0\,.\,(\{[(\mathsf{proj}(i,n)\ x_0)\ /\ x_i]\}_{\circ}^{i\in 1..n}\ (\{e_i\}_{,}^{i\in 1..n}))) \\ \quad \textbf{of}\ (\{x_i\}_{,}^{i\in 1..n}) \rightarrow e_0\end{array}} \quad \text{(r-let)}$$

Figure 2.18: Reduction

$$\boxed{\vdash^{match} p \leftarrow e \rightsquigarrow \varphi}$$

$$\frac{\text{not whnf}(e) \qquad \vdash e \rightarrowtail e' \qquad \vdash^{match} x \leftarrow e' \rightsquigarrow \varphi}{\vdash^{match} x \leftarrow e \rightsquigarrow \varphi} \quad \text{(m-reduce)}$$

$$\frac{\text{whnf}(e)}{\vdash^{match} x \leftarrow e \rightsquigarrow [e \,/\, x]} \quad \text{(m-var)}$$

$$\frac{\{\vdash^{match} p_i \leftarrow e_i \rightsquigarrow \varphi_i\}^{i\in 1..n} \qquad \varphi \equiv \{\varphi_i\}_{\circ}^{i\in 1..n}}{\vdash^{match} (C\ \{p_i\}^{i\in 1..n}) \leftarrow (C\ \{e_i\}^{i\in 1..n}) \rightsquigarrow \varphi} \quad \text{(m-con-1)}$$

$$\frac{C_1 \not\equiv C_2}{\vdash^{match} (C_1\ \{p_i\}^{i\in 1..n}) \leftarrow (C_2\ \{e_i\}^{i\in 1..n}) \rightsquigarrow \bot} \quad \text{(m-con-2)}$$

$$\frac{}{\vdash^{match} (C\ \{p_i\}^{i\in 1..n}) \leftarrow \lambda x\,.\,e \rightsquigarrow \bot} \quad \text{(m-con-3)}$$

$$\frac{}{\vdash^{match} (C\ \{p_i\}^{i\in 1..n}) \leftarrow \bot \rightsquigarrow \bot} \quad \text{(m-con-4)}$$

Figure 2.19: Pattern matching

$$\vdash e_1 \rightarrowtail e_2,$$

meaning that an expression e_1 evaluates in one step to an expression e_2.

If a constructor is applied to one or more arguments, these arguments are reduced one by one, from left to right.

Application of a lambda abstraction results in beta reduction; applying $\bot$ yields failure. The evaluation of applications is lazy, i.e., functions are reduced before their arguments.

The reduction of **case** expressions involves pattern matching. The rules for pattern matching are shown in Figure 2.19 and take the form

$$\vdash^{match} p \leftarrow e \rightsquigarrow \varphi,$$

i.e., matching a pattern p against an expression e produces a substitution φ. Before a pattern is matched against an expression, the expression is reduced to weak-head normal form (see Figure 2.20). Matching variables in patterns simply results in substitutions; constructor applications are matched pointwise. Mismatches are reported by means of the special substitution $\bot$. Notice that if a substitution in composed with $\bot$, the result is, essentially, also $\bot$.

A **case** expression reduces to the expression that is associated with the first pattern that yields a non-failing substitution. If no such pattern exists, reducing the **case** expression results in run-time failure.

Fixed-point applications $(\textbf{fix}\ e)$ are reduced to recursive calls $(e\ (\textbf{fix}\ e))$.

$$\boxed{\mathsf{whnf}(e)}$$

$$\frac{}{\mathsf{whnf}(C\ \{e_i\}^{i\in 1..n})}\ \text{(whnf-con)} \qquad \frac{}{\mathsf{whnf}(\lambda x\,.\,e)}\ \text{(whnf-abs)}$$

$$\frac{}{\mathsf{whnf}(\bot)}\ \text{(whnf-bot)}$$

Figure 2.20: Weak-head normal form

$$\boxed{\mathsf{proj}(m,n) \equiv e}$$

$$\frac{}{\mathsf{proj}(m,n) \equiv \lambda x_0\,.\ \mathbf{case}\ x_0\ \mathbf{of}\ (\{x_i\}_{,}^{i\in 1..n}) \to x_m}\ \text{(proj)}$$

Figure 2.21: Projection functions

The reduction of **let** expressions is essentially a translation in terms of substitutions, tuples and fixed points. It makes use of the metafunction proj for generating projection functions. This metafunction is shown in Figure 2.21; it is defined such that $\mathsf{proj}(m,n)$ produces the function that projects the mth component of an n-tuple.

If well-formedness of programs is established under an initial type environment containing bindings for predefined functions, as opposed to predefined data constructors, the reduction system may need to be extendend with rules for these functions. Alternatively, the set of values can be enlarged.

The evaluation function, depicted in Figure 2.22 from expressions e to values v,

$$\vdash e \rightarrowtail\!\!\!\!\!\rightarrow v,$$

is defined as a sequence of reduction steps; the result of a program is given by the evaluation of its main expression.

$$\boxed{\vdash e \rightarrowtail\!\!\!\!\!\rightarrow v}$$

$$\frac{\{e_{i-1} \rightarrowtail e_i\}^{i\in 1..n} \qquad e_n \equiv v_n}{\vdash e_0 \rightarrowtail\!\!\!\!\!\rightarrow v_n}\ \text{(e-eval)}$$

Figure 2.22: Evaluation

Soundness

The soundness of the type system of the core language with respect to its semantics follows from the usual progress and preservation theorems.

Theorem 2.1 (Progress). If $\mathrm{K}; \Gamma \vdash e :: t$ and $\mathsf{fev}(e) \equiv \varepsilon$, then either e is a value or there is an e' with $\vdash e \rightarrowtail e'$. □

Theorem 2.2 (Preservation). If $\mathrm{K}; \Gamma \vdash e :: t$ and $\vdash e \rightarrowtail e'$, then $\mathrm{K}; \Gamma \vdash e' :: t$. □

2.4 Haskell 98 and Beyond

The core language that is discussed in the previous sections is an excellent vehicle for the formal treatment of issues related to functional programming; throughout this thesis it is used in definitions and algorithms. For the informal parts of this thesis, however, the language of choice is Haskell. Still, we deviate from the Haskell 98 standard in several ways.

Kind Annotations

Kinds do not appear at the surface of a Haskell 98 program; for each type variable a kind is inferred. If a kind cannot be fully determined, $\star$ is assumed as default. Sometimes it is convenient to have more control on the kinds of type variables. Therefore we explicitly annotate each type variable with its kind.

Consider, for instance the following type declaration [45]:

$$\textbf{data}\ \mathrm{Set}\ \mathrm{c}\ \mathrm{a} = \mathit{Set}\ [\mathrm{a}] \mid \mathit{Unused}\ (\mathrm{c}\ \mathrm{a} \to ()).$$

The only purpose of the data constructor *Unused* is to force the kind of c to $\star \to \star$. Explicit kind annotations allow us to write

$$\textbf{data}\ \mathrm{Set}\ (\mathrm{c} :: \star \to \star)\ (\mathrm{a} :: \star) = \mathit{Set}\ [\mathrm{a}]$$

instead.

Using kind annotations, the Haskell representations for the type declarations in Section 2.1 become

$$\begin{array}{ll}
\textbf{data}\ \mathrm{Bool} & = \mathit{False} \mid \mathit{True} \\
\textbf{data}\ \mathrm{Fork}\ (\mathrm{a} :: \star) & = \mathit{Fork}\ \mathrm{a}\ \mathrm{a} \\
\textbf{data}\ \mathrm{List}\ (\mathrm{a} :: \star) & = \mathit{Nil} \mid \mathit{Cons}\ \mathrm{a}\ (\mathrm{List}\ \mathrm{a}) \\
\textbf{data}\ \mathrm{Rose}\ (\mathrm{a} :: \star) & = \mathit{Branch}\ \mathrm{a}\ (\mathrm{List}\ (\mathrm{Rose}\ \mathrm{a})) \\
\textbf{data}\ \mathrm{GRose}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star) & = \mathit{GBranch}\ \mathrm{a}\ (\mathrm{f}\ (\mathrm{GRose}\ \mathrm{f}\ \mathrm{a})) \\
\textbf{data}\ \mathrm{Tree}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star) & = \mathit{Tip}\ \mathrm{a} \mid \mathit{Node}\ (\mathrm{Tree}\ \mathrm{a}\ \mathrm{b})\ \mathrm{b}\ (\mathrm{Tree}\ \mathrm{a}\ \mathrm{b}) \\
\textbf{data}\ \mathrm{Perfect}\ (\mathrm{a} :: \star) & = \mathit{ZeroP} \mid \mathit{SuccP}\ \mathrm{a}\ (\mathrm{Perfect}\ (\mathrm{Fork}\ \mathrm{a})).
\end{array}$$

Note that for lists, Haskell provides the built-in type $[\,]$,

$$\textbf{data}\ [\,]\ (\mathrm{a} :: \star) = [\,] \mid (:)\ \mathrm{a}\ ([\,]\ \mathrm{a}),$$

and that we write $[\mathrm{a}]$ for $[\,]\ \mathrm{a}$, $a : as$ for $((:)\ a\ as)$, and $[\{a_i\}_{,}^{i \in 1..n}]$ for $\{a_i :\}^{i \in 1..n}\ [\,]$. Hence, $(:)$ associates to the right.

$$\boxed{\mathsf{rank}(t) \equiv n}$$

$$\frac{}{\mathsf{rank}(a) \equiv 0} \quad \text{(rank-var)} \qquad \frac{}{\mathsf{rank}(T) \equiv 0} \quad \text{(rank-con)}$$

$$\frac{\mathsf{rank}(t_1) \equiv 0 \qquad \mathsf{rank}(t_2) \equiv n}{\mathsf{rank}(t_1 \to t_2) \equiv n} \quad \text{(rank-fun-1)}$$

$$\frac{\mathsf{rank}(t_1) \equiv m \qquad m \in 1\,.\,. \qquad \mathsf{rank}(t_2) \equiv n}{\mathsf{rank}(t_1 \to t_2) \equiv \max\{m+1, n\}} \quad \text{(rank-fun-2)}$$

$$\frac{t_1 \not\equiv (\to)\ t, \text{for all } t \qquad \mathsf{rank}(t_1) \equiv m \qquad \mathsf{rank}(t_2) \equiv n}{\mathsf{rank}(t_1\ t_2) \equiv \max\{m, n\}} \quad \text{(rank-app)}$$

$$\frac{\mathsf{rank}(t) \equiv n}{\mathsf{rank}(\forall a :: \kappa\ .\ t) \equiv \max\{1, n\}} \quad \text{(rank-forall)}$$

Figure 2.23: Rank of a type

Constructorless Types

The syntax of the core language allows type declarations that do not introduce data constructors; for instance,

data Zero $=$.

Here, Zero is a constructorless type. Besides the diverging value $\bot$, it does not contain any elements.

Declarations of constructorless types are not valid in Haskell 98. However, they are supported by the most popular implementation of the language—the Glasgow Haskell Compiler [26], or GHC for short.

Constructorless types are put to use in Chapter 6.

Arbitrary-rank Polymorphism

In contrast to our core language, Haskell has a predicative type system: Type parameters in Haskell only range over monomorphic types. However, most implementations loosen this restriction for the first parameter of the function-space constructor by supporting rank-2 polymorphism; GHC even supports arbitrary-rank polymorphism [76]. The rank of a type is given by the meta-function rank in Figure 2.23.

In a rank-n system, all types are required to have ranks no higher than n. In this thesis, we assume that there are no restrictions on the rank of a type, i.e., we have arbitrary-rank polymorphism.

The following declaration is an example of a rank-2 typed function:

$$f\ g :: (\forall(\mathrm{a} :: \star)\ .\ [\mathrm{a}] \to [\mathrm{a}]) \to ([\mathrm{Integer}], [\mathrm{Char}])$$
$$f\ g = (g\ [2,3,5], g\ [\texttt{'a'},\texttt{'e'},\texttt{'i'}]).$$

The function f takes a polymorphic function g as argument and applies it to two lists of different types. To be able to write higher-ranked types, we need explicit quantifiers, which are not part of Haskell 98. We also use these to write normal, i.e. rank-1, polymorphic types.

Type inference for systems with rank 3 or higher is undecidable [53]. Therefore, we are required to decorate functions with a rank higher than 2 with explicit type signatures. This poses no real problem, since we already write type signatures for all top-level function declarations for documentation purposes.

Pattern Matching

The pattern matching rules for the core language are a little different from those in Haskell. The following expression, for example,

$$\textbf{case}\ \bot\ \textbf{of}\ \{\mathit{Fork}\ b_1\ b_2 \to b_1; \mathit{bs} \to \mathit{False}\}$$

evaluates in the core language to *False*, while in Haskell it yields $\bot$.

Furthermore, Haskell has special pattern-matching rules for types that are declared by means of the **newtype** construct. Since there is no such construct in the core language, we are not able to adopt these special semantics.

Fortunately, none of these departures is essential to our discussion of functional generic programming.

2.5 Notes

As mentioned, our core language, which can be seen as a variant of the polymorphic lambda calculus [24, 25, 80], is largely inspired by the language that is used by Löh [58]. Proofs for progress and preservation, i.e. Theorems 2.1 and 2.2, can be found there.

The idea of studying a richer language through a smaller, base language is not new [57] and has been proven quite successful. In the Haskell 98 Report [75], for instance, top-level language constructs are translated to a core language, much like the one that appears in this chapter. A similar language is used as the internal language of the Glasgow Haskell Compiler [74].

Chapter 3

Type-indexed Functions

In this chapter, we extend Haskell with the notion of type-indexed functions, basicly a language feature that supports overloading. Type-indexed functions are essential to Generic Haskell; in Chapter 4, they are used to define generic functions.

Here, it is shown how type-indexed functions permit overloading (Section 3.1) and how dependencies between type-indexed functions arise (Section 3.2).

3.1 Overloading

Consider the following three functions:

$$
\begin{array}{ll}
\mathit{plusInteger} & :: \text{Integer} \to \text{Integer} \to \text{Integer} \\
\mathit{plusInteger}\ n_1\ n_2 & = n_1 + n_2 \\
\\
\mathit{plusChar} & :: \text{Char} \to \text{Char} \to \text{Char} \\
\mathit{plusChar}\ c_1\ c_2 & = \mathit{chr}\ (\mathit{ord}\ c_1 + \mathit{ord}\ c_2) \\
\\
\mathit{plusBool} & :: \text{Bool} \to \text{Bool} \to \text{Bool} \\
\mathit{plusBool}\ b_1\ b_2 & = b_1 \vee b_2.
\end{array}
$$

These define addition operations for integers, characters, and Booleans. For integers, we simply use the operator $(+)$; for characters, we add the numerical codes of the arguments to produce the code for the resulting character; and, for Booleans, we take the logical disjunction of the arguments.

Type Indexes

Type-indexed functions allow us to write a single function *plus* that behaves differently depending on the context it is used in; the context is determined by an explicit type argument. We say *plus* is *overloaded*.

$$
\begin{array}{ll}
\mathit{plus}\ \langle\text{Integer}\rangle\ n_1\ n_2 & = n_1 + n_2 \\
\mathit{plus}\ \langle\text{Char}\rangle\ c_1\ c_2 & = \mathit{chr}\ (\mathit{ord}\ c_1 + \mathit{ord}\ c_2) \\
\mathit{plus}\ \langle\text{Bool}\rangle\ b_1\ b_2 & = b_1 \vee b_2
\end{array}
$$

Here, *plus* takes three arguments. The first parameter is special, for it ranges over types instead of values; it is called a *type index*. Type indexes are easily distinguished from other parameters, since they appear within special brackets, $\langle\cdot\rangle$. A function can have one type index at most, and if it has one, it is required to appear in the first argument position.

The types of the second and third argument of *plus* depend on the type that is supplied for the type index. If Integer is used for the type index, then the second and third argument are required to be of type Integer; if we pass Char for the type index, then the second and third argument are required to be of type Char; likewise, if Bool is substituted for the type index, the second and third argument are to be of type Bool. Furthermore, the result type of *plus* is also determined by the type index. For instance,

$$plus\ \langle \text{Integer} \rangle\ 2\ 3$$

evaluates to 5;

$$plus\ \langle \text{Bool} \rangle\ False\ True$$

yields *True*.

By themselves, type-indexed functions make a very modest language extension. The definition of *plus*, above, for example, does nothing more than introducing three individual functions, i.e., *plus* $\langle$Integer$\rangle$, *plus* $\langle$Char$\rangle$, and *plus* $\langle$Bool$\rangle$, that happen to share the same name. In Chapter 4, however, it is shown how type-indexed functions form the backbone of a far more spectacular extension: generic functions.

Types of Type-indexed Functions

It is clear how the functions *plusInteger*, *plusChar*, and *plusBool*, that appear at the start of this section, are related to the type-indexed function *plus*. Given the definition of *plus*, we can now write these functions as

$$\begin{aligned} plusInteger &:: \text{Integer} \to \text{Integer} \to \text{Integer} \\ plusInteger &= plus\ \langle \text{Integer} \rangle \\ \\ plusChar &:: \text{Char} \to \text{Char} \to \text{Char} \\ plusChar &= plus\ \langle \text{Char} \rangle \\ \\ plusBool &:: \text{Bool} \to \text{Bool} \to \text{Bool} \\ plusBool &= plus\ \langle \text{Bool} \rangle. \end{aligned}$$

This implies that

$$\begin{aligned} plus\ \langle \text{Integer} \rangle &:: \text{Integer} \to \text{Integer} \to \text{Integer} \\ plus\ \langle \text{Char} \rangle &:: \text{Char} \to \text{Char} \to \text{Char} \\ plus\ \langle \text{Bool} \rangle &:: \text{Bool} \to \text{Bool} \to \text{Bool}. \end{aligned}$$

So, applying a type-indexed function to a type yields a normally typed function.

It remains to assign a type to an unapplied type-indexed function. Such a type should reflect the fact that the first parameter of a type-indexed function is indeed a type index. For instance, the type of *plus* is given by

$$plus :: \langle a :: \star \rangle \to a \to a \to a,$$

expressing that the first parameter of *plus* is a type index that ranges over types of kind $\star$. The type signature of *plus* reveals how the types of the second and third argument, as well as the result type, depend on the type index. A type index is often written at the left-hand side of a type signature:

$$plus\ \langle a :: \star \rangle :: a \to a \to a.$$

Signatures of Type-indexed Functions

The type of a type-indexed function does not tell the whole story. For example, although the type of *plus* mentions a $\star$-kinded type index, it is not allowed to pass to *plus* just any type that has kind $\star$. More precise, it is not allowed to pass a type other than Integer, Char, or Bool. If we call *plus* with, for instance, the type argument Float, as in

$$plus\ \langle \text{Float} \rangle\ 2.35\ 7.9,$$

we yield a compile-time error, because the definiton of *plus* does not provide a case for Float.

We say that Integer, Char, and Bool, for which the definition of *plus* does define cases, form the *signature* of *plus*. (Not to be confused with the type signature of *plus*, which we often simply refer to as the type of *plus*.) The compile-time error that occurs when a type-indexed function is called with a type that does not match with one of the type patterns appearing in its signature, is called a *specialization error*.

Together, the type and the signature of a type-indexed function determine which types and which values can be passed as arguments to the function.

3.2 Dependencies

Recall that a function can have at most one type index. Besides that, it can have any number of 'normal', value parameters. It is, for example, perfectly legal to have a type-indexed function that takes no value parameters, as in

$$\begin{aligned} &zero\ \langle a :: \star \rangle &&:: a \\ &zero\ \langle \text{Integer} \rangle &&= 0 \\ &zero\ \langle \text{Char} \rangle &&= \texttt{'\textbackslash NUL'} \\ &zero\ \langle \text{Bool} \rangle &&= \mathit{False}. \end{aligned}$$

In mathematics, a *monoid* $(M, \oplus, e)$ is a set M, equipped with an associative, binary operation $\oplus : M \times M \to M$ and an element $e \in M$, such that $e \oplus x = x$ and $x \oplus e = x$ for all $x \in M$. If we represent monoids with the Haskell type Monoid,

$$\textbf{data}\ \text{Monoid}\ (a :: \star) = \mathit{Monoid}\ (a \to a \to a)\ a,$$

then the following type-indexed function can be used to construct monoids for integers, characters, and Booleans:

$$\begin{array}{ll} \mathit{monoid}\ \langle \mathrm{a} :: \star \rangle & :: \mathrm{Monoid\ a} \\ \mathit{monoid}\ \langle \mathrm{Integer} \rangle & = \mathit{Monoid}\ (\mathit{plus}\ \langle \mathrm{Integer} \rangle)\ (\mathit{zero}\ \langle \mathrm{Integer} \rangle) \\ \mathit{monoid}\ \langle \mathrm{Char} \rangle & = \mathit{Monoid}\ (\mathit{plus}\ \langle \mathrm{Char} \rangle)\ (\mathit{zero}\ \langle \mathrm{Char} \rangle) \\ \mathit{monoid}\ \langle \mathrm{Bool} \rangle & = \mathit{Monoid}\ (\mathit{plus}\ \langle \mathrm{Bool} \rangle)\ (\mathit{zero}\ \langle \mathrm{Bool} \rangle). \end{array}$$

Note that calls to the type-indexed functions *plus* and *zero* appear at the right-hand side of this definition.

Parameterized Type Patterns

Lists form a monoid as well: they have concatenation, $(+\!\!+)$, as distinguished operation and the empty list, $[\,]$, as neutral element. However, we cannot just add a case for the list type $[\,]$ to the type-indexed function *monoid*, since $[\,]$ has kind $\star \to \star$ and the type of *monoid* states that it is indexed by types of kind $\star$.

Therefore, we have to apply $[\,]$ to a type of kind $\star$ to yield a type that can be used as a type pattern for *monoid*. For instance:

$$\mathit{monoid}\ \langle [\mathrm{Integer}] \rangle = \mathit{Monoid}\ (\mathit{plus}\ \langle [\mathrm{Integer}] \rangle)\ (\mathit{zero}\ \langle [\mathrm{Integer}] \rangle),$$

where the corresponing cases for *plus* and *zero* are given by

$$\begin{array}{ll} \mathit{plus}\ \langle [\mathrm{Integer}] \rangle\ \mathit{ns}_1\ \mathit{ns}_2 & = \mathit{ns}_1 +\!\!+ \mathit{ns}_2 \\ \mathit{zero}\ \langle [\mathrm{Integer}] \rangle & = [\,]. \end{array}$$

But clearly, the types $[\mathrm{Char}]$, $[\mathrm{Bool}]$, and all other well-kinded applications of $[\,]$ constitute monoids too. We could easily add cases for these types to *plus*, *zero*, and *monoid*, but these would only differ from the cases for $[\mathrm{Integer}]$ in the type indexes. Hence, we rather abstract over the type of list elements and write

$$\begin{array}{ll} \mathit{plus}\ \langle [\alpha] \rangle\ \mathit{as}_1\ \mathit{as}_2 & = \mathit{as}_1 +\!\!+ \mathit{as}_2 \\ \mathit{zero}\ \langle [\alpha] \rangle & = [\,] \\ \mathit{monoid}\ \langle [\alpha] \rangle & = \mathit{Monoid}\ (\mathit{plus}\ \langle [\alpha] \rangle)\ (\mathit{zero}\ \langle [\alpha] \rangle). \end{array}$$

Here, α is a so-called *dependency variable*. It acts as a placeholder for any type that makes the type patterns of *plus*, *zero*, and *monoid* well-kinded, i.e., for any type that has kind $\star$. For example, if the type-indexed function *monoid* is called with the type argument $[\mathrm{Char}]$, the type constructor Char is matched against the dependency variable α, reducing the call to

$$\mathit{Monoid}\ (\mathit{plus}\ \langle [\mathrm{Char}] \rangle)\ (\mathit{zero}\ \langle [\mathrm{Char}] \rangle).$$

The calls $\mathit{plus}\ \langle [\mathrm{Char}] \rangle$ and $\mathit{zero}\ \langle [\mathrm{Char}] \rangle$ then match against, respectively, the cases $\mathit{plus}\ \langle [\alpha] \rangle$ and $\mathit{zero}\ \langle [\alpha] \rangle$, so we yield

$$\mathit{Monoid}\ (+\!\!+)\ [\,].$$

Dependency Constraints

Abstracting over a dependency variable may give rise to dependencies.

Consider, for example, the data type Maybe of optional values,

$$\textbf{data}\ \mathrm{Maybe}\ (\mathrm{a} :: \star) = \mathit{Nothing} \mid \mathit{Just}\ \mathrm{a}.$$

We define addition of optional values as

$$\begin{aligned}
&\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle\ \mathit{Nothing}\ \mathit{Nothing} &&= \mathit{Nothing}\\
&\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle\ \mathit{Nothing}\ (\mathit{Just}\ a_2) &&= \mathit{Just}\ a_2\\
&\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle\ (\mathit{Just}\ a_1)\ \mathit{Nothing} &&= \mathit{Just}\ a_1\\
&\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle\ (\mathit{Just}\ a_1)\ (\mathit{Just}\ a_2) &&= \mathit{Just}\ (\mathit{plus}\ \langle\alpha\rangle\ a_1\ a_2),
\end{aligned}$$

and, consequently,

$$\begin{aligned}
&\mathit{zero}\ \langle \text{Maybe}\ \alpha\rangle &&= \mathit{Nothing}\\
&\mathit{monoid}\ \langle \text{Maybe}\ \alpha\rangle &&= \mathit{Monoid}\ (\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle)\ (\mathit{zero}\ \langle \text{Maybe}\ \alpha\rangle).
\end{aligned}$$

But the last case for $\mathit{plus}\ \langle \text{Maybe}\ \alpha\rangle$ only makes sense if *plus* is also defined for the type represented by α. For instance, it is perfectly legal to call *plus* with the type argument Maybe Integer, for Integer is in the signature of *plus*; so

$$\mathit{plus}\ \langle \text{Maybe Integer}\rangle\ (\mathit{Just}\ 2)\ (\mathit{Just}\ 3)$$

evaluates to *Just* 5. On the other hand, calling *plus* with the type constructor Float as type argument is not allowed, since Float cannot be matched against any of the type patterns appearing in the signature of *plus*. More specific,

$$\mathit{plus}\ \langle \text{Maybe Float}\rangle\ (\mathit{Just}\ 2.35)\ (\mathit{Just}\ 7.9)$$

would, if allowed, reduce to

$$\mathit{Just}\ (\mathit{plus}\ \langle \text{Float}\rangle\ 2.35\ 7.9),$$

which is erroneous.

To reflect this restriction, we adapt the type signature of *plus*:

$$\mathit{plus}\ \langle \text{a} :: \star\rangle :: (\mathit{plus}\ \langle \text{a}\rangle) \Rightarrow \text{a} \to \text{a} \to \text{a}.$$

Here, $(\mathit{plus}\ \langle \text{a}\rangle)$ is a *dependency constraint*. It expresses that $\mathit{plus}\ \langle \text{a} :: \star\rangle$ depends on *plus* in the type variable a.

In general, if the definition of a type-indexed function x_1 contains a call to a type-indexed function x_2 with a type argument that contains a dependency variable, one or more dependencies for x_1 arise: If the dependency variable appears in the left-most position of the type argument, a dependency on x_2 is generated. If the dependency variable appears in another position, dependencies are generated on the functions that x_2 depends on.

A type-indexed function that depends on itself, like *plus*, is called *reflexive*.

Adding the appropriate dependency constraints to the types of *zero* and *monoid* gives

$$\begin{aligned}
&\mathit{zero}\ \langle \text{a} :: \star\rangle &&:: \text{a}\\
&\mathit{monoid}\ \langle \text{a} :: \star\rangle :: (\mathit{plus}\ \langle \text{a}\rangle) \Rightarrow &&\text{Monoid a}.
\end{aligned}$$

The dependency for *monoid* on *plus* arises from the cases for $[\,]$ and Maybe.

3.3 Notes

Overloading is a form of *ad-hoc polymorphism* [16].

Haskell 98 already provides a mechanism for overloading: type classes [90]. Actually, the $(+)$ operator, that is used in the implementation of the type-indexed function *plus*, is overloaded using the type-class system. Type classes are somewhat more flexible than type-indexed functions. In particular, they allow dependencies, in the form of class constraints, to arise per case, i.e., per class instance. Furthermore, type indexes for type-class functions are inferred; there is no need to explicitly provide a type argument for a call to an overloaded function. The reason we present type-indexed functions here, is that in Generic Haskell, as shown in Chapter 4, they are the mechanism on top of which generics are implemented.

Jones [52] points out that, in absence of a single natural implementation for each instance, monoids are not well-suited to overloading. In particular, a monoid structure on integers could as well be given in terms of multiplication.

Dependencies, as discussed in this chapter, form the core of what is called Dependency-style Generic Haskell [59].

Chapter 4

Generics

In the previous chapter we showed how Generic Haskell extends Haskell 98 with the notion of type-indexed functions. On itself, this extension is not a very spectaculair one: it only introduces a limited form of ad-hoc polymorphism. In this chapter, we present a far more compelling reason for adding type-indexed functions to the language: we show how they support the definition of generic functions.

A generic function is a function that is defined for only a small number of data types; yet, it can be applied to a whole class of types. In Generic Haskell, generic functions are defined by induction over the structure of types. The ability to define such functions enriches the language with the notion of *structural polymorphism* [81, 82].

The chapter is organized as follows. Section 4.1 provides two examples of generic algorithms. Section 4.2 shows how, in Generic Haskell, generic algorithms can be captured in terms of type-indexed functions over structural representations of data types. Section 4.3 discusses some extensions of the basic machinery for generic functions.

4.1 Generic Algorithms

In this section, we examine two basic algorithms: one for data compression and one for equality. These algorithms are generic in the sense that they can be applied to a broad range of data types. However, we still need to provide a suitable instance for each individual data type. In Section 4.2, we show how Generic Haskell permits true generic implementations of these algorithms.

Data Compression

Our first example of a generic algorithm involves data compression [47]. The objective is to derive a binary encoding for the elements of the data types that appeared in Chapter 2. For convenience, their definitions are repeated here:

```
data Bool                = False | True
data Fork (a :: ⋆)       = Fork a a
```

```
data [a :: ⋆]                          = [] | a : [a]
data Rose (a :: ⋆)                     = Branch a [Rose a]
data GRose (f :: ⋆ → ⋆) (a :: ⋆)       = GBranch a (f (GRose f a))
data Tree (a :: ⋆) (b :: ⋆)            = Tip a | Node (Tree a b) b (Tree a b)
data Perfect (a :: ⋆)                  = ZeroP | SuccP a (Perfect (Fork a)).
```

For our purposes, a binary encoding is just a list of bits:

```
data Bit  = 0 | 1
type Bin = [Bit].
```

Note that we use `0` and `1` as data constructors, to simplify the presentation.

We start by defining a function that encodes Boolean values:

```
encodeBool       :: Bool → Bin
encodeBool False = 0 : []
encodeBool True  = 1 : [].
```

The value of type Bool is encoded by a single bit: `0` for *False*, `1` for *True*. Given this definition of *encodeBool*, it is fairly easy to construct a decoder for Booleans:

```
decodesBool          :: Bin → (Bool, Bin)
decodesBool []       = error "decodesBool"
decodesBool (0 : bin) = (False, bin)
decodesBool (1 : bin) = (True, bin).
```

To facilitate composition of decoders, *decodesBool* takes a list of bits[1] and produces a pair, consisting of a decoded Bool value and the remaining part of the bit sequence. For instance, *decodesBool* $[0,0,1]$ evaluates to $(False, [0,1])$.

The encoder for Booleans can be used to construct a function *encodeForkBool*, that encodes values of type Fork Bool:

```
encodeForkBool                :: Fork Bool → Bin
encodeForkBool (Fork b1 b2) = encodeBool b1 ++ encodeBool b2.
```

A value *Fork* b_1 b_2 is encoded by concatenating the encodings of b_1 and b_2. For example, *Fork False True* is encoded as $[0,1]$. Likewise, the decoder for Booleans is used to implement the function *decodesForkBool*:

```
decodesForkBool       :: Bin → (Fork Bool, Bin)
decodesForkBool bin = let (b1, bin')  = decodesBool bin
                          (b2, bin'') = decodesBool bin'
                      in  (Fork b1 b2, bin'').
```

Abstracting over Bool gives a more general encoder-decoder pair:

```
encodeFork                  :: ∀(a :: ⋆) . (a → Bin) → Fork a → Bin
encodeFork ena (Fork a1 a2) = ena a1 ++ ena a2
```

[1] Notice that the decoder diverges on incorrect input. An alternative approach to dealing with incorrect input would be to let *decodesBool* produce a list of possible decodings [88]. Incorrect input then results in the empty list, whereas correct input yields a singleton list.

$$
\begin{array}{l}
\mathit{decodesFork} \quad :: \\
\quad \forall(\mathsf{a} :: \star)\ .\ (\mathsf{Bin} \to (\mathsf{a}, \mathsf{Bin})) \to \mathsf{Bin} \to (\mathsf{Fork\ a}, \mathsf{Bin}) \\
\mathit{decodesFork\ dea\ bin} \quad = \mathbf{let}\ (a_1, bin') \ = \mathit{dea\ bin} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ (a_2, bin'') = \mathit{dea\ bin'} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (\mathit{Fork}\ a_1\ a_2, bin'').
\end{array}
$$

Compared to *encodeForkBool* and *decodesForkBool*, *encodeFork* and *decodesFork* both take an extra argument function for, respectively, encoding and decoding values of the argument type a.

This abstraction step is much like the step taken in Section 3.2 to get from the definition of *plus* $\langle[\mathsf{Integer}]\rangle$ to *plus* $\langle[\alpha]\rangle$. Indeed, the functions *encodeBool* and *encodeFork* can be assimilated into a type-indexed function *encode*:

$$
\begin{array}{l}
\mathit{encode}\ \langle \mathsf{a} :: \star \rangle \quad\quad\quad\quad\quad\quad :: (\mathit{encode}\ \langle \mathsf{a} \rangle) \Rightarrow \mathsf{a} \to \mathsf{Bin} \\
\mathit{encode}\ \langle \mathsf{Bool} \rangle\ \mathit{False} \quad\quad\quad\quad = \mathtt{0} \\
\mathit{encode}\ \langle \mathsf{Bool} \rangle\ \mathit{True} \quad\quad\quad\quad\ = \mathtt{1} \\
\mathit{encode}\ \langle \mathsf{Fork}\ \alpha \rangle\ (\mathit{Fork}\ a_1\ a_2) = \mathit{encode}\ \langle \alpha \rangle\ a_1 +\!\!+ \mathit{encode}\ \langle \alpha \rangle\ a_2.
\end{array}
$$

Note how the universal quantification in the type of *encodeFork* corresponds to the dependency variable in the case for *encode* $\langle \mathsf{Fork}\ \alpha \rangle$. Furthermore, there is a connection between the extra parameter *ena* of *encodeFork* and the dependency constraint $(\mathit{encode}\ \langle \mathsf{a} \rangle)$ for *encode* $\langle \mathsf{a} :: \star \rangle$. Actually, the semantics of dependencies between type-indexed functions are given in terms of parametric polymorphism and function arguments [37, 39, 59, 58]; hence, the Generic Haskell compiler would translate *encode* $\langle \mathsf{a} :: \star \rangle$ into a pair of Haskell functions that closely resemble *encodeBool* and *encodeFork* [92, 17, 18, 60]. A type-indexed decoder is given by

$$
\begin{array}{l}
\mathit{decodes}\ \langle \mathsf{a} :: \star \rangle \quad\quad\quad\quad :: (\mathit{decodes}\ \langle \mathsf{a} \rangle) \Rightarrow \mathsf{Bin} \to (\mathsf{a}, \mathsf{Bin}) \\
\mathit{decodes}\ \langle \mathsf{Bool} \rangle\ [\,] \quad\quad\quad\ = \mathit{error}\ \texttt{"decodes"} \\
\mathit{decodes}\ \langle \mathsf{Bool} \rangle\ (\mathtt{0} : \mathit{bin}) = (\mathit{False}, \mathit{bin}) \\
\mathit{decodes}\ \langle \mathsf{Bool} \rangle\ (\mathtt{1} : \mathit{bin}) = (\mathit{True}, \mathit{bin}) \\
\mathit{decodes}\ \langle \mathsf{Fork}\ \alpha \rangle\ \mathit{bin} \quad = \mathbf{let}\ (a_1, bin') \ = \mathit{decodes}\ \langle \alpha \rangle\ \mathit{bin} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ (a_2, bin'') = \mathit{decodes}\ \langle \alpha \rangle\ \mathit{bin'} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (\mathit{Fork}\ a_1\ a_2, bin'').
\end{array}
$$

Now let us extend the type-indexed *encode* function with cases for lists:

$$
\begin{array}{l}
\mathit{encode}\ \langle [\alpha] \rangle\ [\,] \quad\ = \mathtt{0} : [\,] \\
\mathit{encode}\ \langle [\alpha] \rangle\ (a : as) = \mathtt{1} : \mathit{encode}\ \langle \alpha \rangle\ a +\!\!+ \mathit{encode}\ \langle [\alpha] \rangle\ as.
\end{array}
$$

One bit is emitted to distinguish between the data constructors $[\,]$ and $(:)$. For the empty list, we just produce $[\mathtt{0}]$. For a non-empty list $(a : as)$, we emit $\mathtt{1}$, followed by the encoding of the first element a and the recursive encoding of the tail list as. The corresponding cases for *decodes* are

$$
\begin{array}{l}
\mathit{decodes}\ \langle [\alpha] \rangle\ [\,] \quad\ = \mathit{error}\ \texttt{"decodes"} \\
\mathit{decodes}\ \langle [\alpha] \rangle\ (\mathtt{0} : \mathit{bin}) = ([\,], \mathit{bin}) \\
\mathit{decodes}\ \langle [\alpha] \rangle\ (\mathtt{1} : \mathit{bin}) = \mathbf{let}\ (a, bin') \ = \mathit{decodes}\ \langle \alpha \rangle\ \mathit{bin} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ (as, bin'') = \mathit{decodes}\ \langle [\alpha] \rangle\ \mathit{bin'} \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in}\ \ (a : as, bin'').
\end{array}
$$

These are all instances of the same general scheme: every choice between two data constructors is encoded by a single bit; the fields of a constructor are recursively encoded, after which the resulting bit sequences are concatenated. Following this scheme, it is easy to define encoders and decoders for rose trees,

$$
\begin{aligned}
&\mathit{encode}\ \langle \text{Rose}\ \alpha\rangle\ (\mathit{Branch}\ a\ as) = \mathit{encode}\ \langle\alpha\rangle\ a +\!\!+ \mathit{encode}\ \langle[\text{Rose}\ \alpha]\rangle\ as\\
&\mathit{decodes}\ \langle \text{Rose}\ \alpha\rangle\ bin \quad = \mathbf{let}\ (a, bin') \quad = \mathit{decodes}\ \langle\alpha\rangle\ bin\\
&\qquad\qquad\qquad\qquad\quad\ \ (as, bin'') = \mathit{decodes}\ \langle[\text{Rose}\ \alpha]\rangle\ bin'\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ (\mathit{Branch}\ a\ as, bin''),
\end{aligned}
$$

generalized rose trees,

$$
\begin{aligned}
&\mathit{encode}\ \langle \text{GRose}\ \varphi\ \alpha\rangle\ (\mathit{GBranch}\ a\ as) =\\
&\quad \mathit{encode}\ \langle\alpha\rangle\ a +\!\!+ \mathit{encode}\ \langle\varphi\ (\text{GRose}\ \varphi\ \alpha)\rangle\ as\\
&\mathit{decodes}\ \langle \text{GRose}\ \varphi\ \alpha\rangle\ bin \quad =\\
&\quad \mathbf{let}\ (a, bin') \quad = \mathit{decodes}\ \langle\alpha\rangle\ bin\\
&\qquad\ \ (as, bin'') = \mathit{decodes}\ \langle\varphi\ (\text{GRose}\ \varphi\ \alpha)\rangle\ bin'\\
&\quad \mathbf{in}\ \ (\mathit{GBranch}\ a\ as, bin''),
\end{aligned}
$$

external binary search-trees,

$$
\begin{aligned}
&\mathit{encode}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ (\mathit{Tip}\ a) \quad = \mathtt{0} : \mathit{encode}\ \langle\alpha\rangle\ a\\
&\mathit{encode}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ (\mathit{Node}\ l\ b\ r) =\\
&\quad \mathtt{1} : \mathit{encode}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ l +\!\!+ \mathit{encode}\ \langle\beta\rangle\ b +\!\!+ \mathit{encode}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ r\\
&\mathit{decodes}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ [\,] \quad = \mathit{error}\ \texttt{"decodes"}\\
&\mathit{decodes}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ (\mathtt{0} : bin) \quad = \mathbf{let}\ (a, bin') = \mathit{decodes}\ \langle\alpha\rangle\ bin\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ (\mathit{Tip}\ a, bin')\\
&\mathit{decodes}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ (\mathtt{1} : bin) \quad = \mathbf{let}\ (l, bin') \quad = \mathit{decodes}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ bin\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ (b, bin'') \quad = \mathit{decodes}\ \langle\beta\rangle\ bin'\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ (r, bin''') = \mathit{decodes}\ \langle \text{Tree}\ \alpha\ \beta\rangle\ bin''\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ (\mathit{Node}\ l\ b\ r, bin'''),
\end{aligned}
$$

and perfectly balanced, binary leaf trees,

$$
\begin{aligned}
&\mathit{encode}\ \langle \text{Perfect}\ \alpha\rangle\ \mathit{ZeroP} \quad = \mathtt{0} : [\,]\\
&\mathit{encode}\ \langle \text{Perfect}\ \alpha\rangle\ (\mathit{SuccP}\ a\ as) =\\
&\quad \mathtt{1} : \mathit{encode}\ \langle\alpha\rangle\ a +\!\!+ \mathit{encode}\ \langle \text{Perfect}\ (\text{Fork}\ \alpha)\rangle\ as\\
&\mathit{decodes}\ \langle \text{Perfect}\ \alpha\rangle\ [\,] \quad = \mathit{error}\ \texttt{"decodes"}\\
&\mathit{decodes}\ \langle \text{Perfect}\ \alpha\rangle\ (\mathtt{0} : bin) \quad = (\mathit{ZeroP}, bin)\\
&\mathit{decodes}\ \langle \text{Perfect}\ \alpha\rangle\ (\mathtt{1} : bin) \quad =\\
&\quad \mathbf{let}\ (a, bin') \quad = \mathit{decodes}\ \langle\alpha\rangle\ bin\\
&\qquad\ \ (as, bin'') = \mathit{decodes}\ \langle \text{Perfect}\ (\text{Fork}\ \alpha)\rangle\ bin'\\
&\quad \mathbf{in}\ \ (\mathit{SuccP}\ a\ as, bin'').
\end{aligned}
$$

Two issues deserve some attention:

First, consider the Haskell translation for $\mathit{encode}\ \langle \text{GRose}\ \varphi\ \alpha\rangle$, which resembles

$$
\begin{aligned}
&\mathit{encodeGRose} \quad ::\\
&\quad \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .
\end{aligned}
$$

$$
\begin{aligned}
&(\forall(\mathsf{b} :: \star)\ .\ (\mathsf{b} \to \mathsf{Bin}) \to \mathsf{f\ b} \to \mathsf{Bin}) \\
&\quad \to (\mathsf{a} \to \mathsf{Bin}) \to \mathsf{GRose\ f\ a} \to \mathsf{Bin} \\
&\mathit{encodeGRose\ enf\ ena}\ (\mathit{GBranch\ a\ as}) = \\
&\quad \mathit{ena\ a} +\!\!+ \mathit{enf}\ (\mathit{encodeGRose\ enf\ ena})\ \mathit{as}.
\end{aligned}
$$

Notice that this function has a polymorphic type of rank 2. It is a common phenomenon that translations of cases that involve higher-kinded[2] dependency variables of give rise to functions with higher-ranked types. Therefore, Generic Haskell requires that its Haskell backend supports arbitrary-rank polymorphism.

Second, the translation of *encode* $\langle \mathsf{Perfect}\ \alpha \rangle$, which is essentially

$$
\begin{aligned}
&\mathit{encodePerfect} && :: \forall(\mathsf{a} :: \star)\ .\ (\mathsf{a} \to \mathsf{Bin}) \to \mathsf{Perfect\ a} \to \mathsf{Bin} \\
&\mathit{encodePerfect\ ena\ ZeroP} && = \mathtt{0} : [\,] \\
&\mathit{encodePerfect\ ena}\ (\mathit{SuccP\ a\ as}) = \\
&\quad \mathtt{1} : \mathit{ena\ a} +\!\!+ \mathit{encodePerfect}\ (\mathit{encodeFork\ ena})\ \mathit{as},
\end{aligned}
$$

employs a special form of recursion, known as *polymorphic recursion* [67], i.e., *encodePerfect* is used polymorphically within its own definition. Type inference in the presence of polymorphic recursion is undecidable [30]; hence, Haskell 98 allows polymorphic-recursive functions, but only if those are decorated with explicit type signatures. When translating type-indexed functions, polymorphic recursion typically arises when a function is reflexive and a nested data type is used as a type index.

Equality

Surely the most popular generic algorithm is equality: it decides whether two given values of the same data type are equal to each other. Here it is implemented by means of a reflexive type-indexed function *eq*,

$$\mathit{eq}\ \langle \mathsf{a} :: \star \rangle :: (\mathit{eq}\ \langle \mathsf{a} \rangle) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

For instance, for equality of Booleans, we have

$$
\begin{aligned}
&\mathit{eq}\ \langle \mathsf{Bool} \rangle\ \mathit{False\ False} = \mathit{True} \\
&\mathit{eq}\ \langle \mathsf{Bool} \rangle\ \mathit{False\ True}\ = \mathit{False} \\
&\mathit{eq}\ \langle \mathsf{Bool} \rangle\ \mathit{True\ False}\ = \mathit{False} \\
&\mathit{eq}\ \langle \mathsf{Bool} \rangle\ \mathit{True\ True}\ = \mathit{True},
\end{aligned}
$$

i.e., two Boolean values are equal if both are *False* or both are *True*.

Equality for Fork values is determined pointwise:

$$\mathit{eq}\ \langle \mathsf{Fork}\ \alpha \rangle\ (\mathit{Fork}\ a_{11}\ a_{12})\ (\mathit{Fork}\ a_{21}\ a_{22}) = \mathit{eq}\ \langle \alpha \rangle\ a_{11}\ a_{21} \wedge \mathit{eq}\ \langle \alpha \rangle\ a_{12}\ a_{22}.$$

Two lists of the same type are equal if they are both empty or if they are both non-empty and have equal heads and tails:

$$
\begin{aligned}
&\mathit{eq}\ \langle [\alpha] \rangle\ [\,]\ [\,] && = \mathit{True} \\
&\mathit{eq}\ \langle [\alpha] \rangle\ [\,]\ (a_2 : as_2) && = \mathit{False}
\end{aligned}
$$

[2] 'Higher-kinded' means 'having a higher-order kind', i.e., a kind of order n, for $n \in 2..$

$$
\begin{aligned}
&eq\ \langle[\alpha]\rangle\ (a_1 : as_1)\ [\,] &&= False\\
&eq\ \langle[\alpha]\rangle\ (a_1 : as_1)\ (a_2 : as_2) &&= eq\ \langle\alpha\rangle\ a_1\ a_2 \wedge eq\ \langle[\alpha]\rangle\ as_1\ as_2.
\end{aligned}
$$

In general, two values are equal if they are built from the same data constructor and they have equal components. Just like the definitions for data compression, definitions of equality closely follow the structure of types. So, for values of types Rose, GRose, Tree, and Perfect, we have

$$
\begin{aligned}
&eq\ \langle\text{Rose}\ \alpha\rangle\ (Branch\ a_1\ as_1)\ (Branch\ a_2\ as_2) &&=\\
&\quad eq\ \langle\alpha\rangle\ a_1\ a_2 \wedge eq\ \langle[\text{Rose}\ \alpha]\rangle\ as_1\ as_2\\
&eq\ \langle\text{GRose}\ \varphi\ \alpha\rangle\ (GBranch\ a_1\ as_1)\ (GBranch\ a_2\ as_2) &&=\\
&\quad eq\ \langle\alpha\rangle\ a_1\ a_2 \wedge eq\ \langle\varphi\ (\text{GRose}\ \varphi\ \alpha)\rangle\ as_1\ as_2\\
&eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ (Tip\ a_1)\ (Tip\ a_2) &&= eq\ \langle\alpha\rangle\ a_1\ a_2\\
&eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ (Tip\ a_1)\ (Node\ l_2\ b_2\ r_2) &&= False\\
&eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ (Node\ l_1\ b_1\ r_1)\ (Tip\ a_2) &&= False\\
&eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ (Node\ l_1\ b_1\ r_1)\ (Node\ l_2\ b_2\ r_2) &&=\\
&\quad eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ l_1\ l_2 \wedge eq\ \langle\beta\rangle\ b_1\ b_2 \wedge eq\ \langle\text{Tree}\ \alpha\ \beta\rangle\ r_1\ r_2\\
&eq\ \langle\text{Perfect}\ \alpha\rangle\ ZeroP\ ZeroP &&= True\\
&eq\ \langle\text{Perfect}\ \alpha\rangle\ ZeroP\ (SuccP\ a_2\ as_2) &&= False\\
&eq\ \langle\text{Perfect}\ \alpha\rangle\ (SuccP\ a_1\ as_1)\ ZeroP &&= False\\
&eq\ \langle\text{Perfect}\ \alpha\rangle\ (SuccP\ a_1\ as_1)\ (SuccP\ a_2\ as_2) &&=\\
&\quad eq\ \langle\alpha\rangle\ a_1\ a_2 \wedge eq\ \langle\text{Perfect}\ (\text{Fork}\ \alpha)\rangle\ as_1\ as_2.
\end{aligned}
$$

In the next section, it is shown how 'following the structure of types' can be captured in the definition of a generic definition.

4.2 Generic Functions

In the previous section, several ad-hoc instances of generic data compression and generic equality were given. These instances all follow general schemes for deriving appropriate cases for *encodes*, *decodes*, and *eq*. If we would decide to extend these functions for an additional data type, it would be crystal clear what the required extra cases should look like. In fact, defining compression and equality for new types is boring and, consequently, error-prone.

Fortunately, Generic Haskell permits generic algorithms like data compression and equality to be fully captured by type-indexed functions that are defined for only a small number of type indexes. These type-indexed functions, illustratively called *generic functions*, can then be applied to a whole class of data types.

In this section, we show that it suffices to cover only three basic data types and a small set of primitive type constructors.

Sums and Products

Essentially, a Haskell data type is a *sum of products*: Each data constructor of a type constitutes a term of a sum. Each field of a constructor makes a factor of a product.

Using the following types,

```
data Unit                 = Unit
data Prod (a :: ⋆) (b :: ⋆) = a × b
data Sum (a :: ⋆) (b :: ⋆) = Inl a | Inr b,
```

to represent nullary products, binary products, and binary sums, we can derive a sum-of-product representation for each algebraic data type.

For instance, the type Bool,

```
data Bool = False | True,
```

has two data constructors and is therefore represented as a binary sum. Both constructors, *False* and *True*, have no fields; so, they are represented as nullary products. Hence, Bool is represented by

```
type Bool° = Sum Unit Unit.
```

Likewise, Fork,

```
data Fork (a :: ⋆) = Fork a a,
```

has a single, binary data constructor and is represented by

```
type Fork° (a :: ⋆) = Prod a a.
```

Recursive types, like [ ],

```
data [a :: ⋆] = [ ] | a : [a],
```

are represented in a non-recursive fashion:

```
type [a :: ⋆]° = Sum Unit (Prod a [a]).
```

So, at the right hand side the type [a] appears, as opposed to [a]°.

For Rose, GRose, Tree, and Perfect,

```
data Rose (a :: ⋆)                 = Branch a [Rose a]
data GRose (f :: ⋆ → ⋆) (a :: ⋆) = GBranch a (f (GRose f a))
data Tree (a :: ⋆) (b :: ⋆)        = Tip a | Node (Tree a b) b (Tree a b)
data Perfect (a :: ⋆)              = ZeroP | SuccP a (Perfect (Fork a)),
```

we have

```
type Rose° (a :: ⋆)                 = Prod a [Rose a]
type GRose° (f :: ⋆ → ⋆) (a :: ⋆) = Prod a (f (GRose f a))
type Tree° (a :: ⋆) (b :: ⋆)        =
  Sum a (Prod (Tree a b) (Prod b (Tree a b)))
type Perfect° (a :: ⋆)              =
  Sum Unit (Prod a (Perfect (Fork a))).
```

Note that the ternary data constructor *Node* of Tree is represented as a nesting of binary products. Likewise, types with more than two constructors, like the type Sequ of binary random-access lists [70],

```
data Sequ (a :: ⋆) = EndS
```

```
  | ZeroS (Sequ (Fork a))
  | OneS a (Sequ (Fork a)),
```

are represented as nested binary sums:

```
type Sequ° (a :: ⋆) =
  Sum Unit (Sum (Sequ (Fork a)) (Prod a (Sequ (Fork a)))).
```

The representation of a data type as a sum of products is called its *structural representation*. The resulting type synonym is called a *structure type*; values of a structure type are called *structure values*.

Going Generic

The significance of structural representations becomes apparent when we consider how generic behaviour of type-indexed functions is derived:

If a type-indexed function is called with a type argument that cannot directly be matched against a type pattern appearing in the signature of the function, then the type argument and the corresponding arguments are converted to their structural representations, after which the type-indexed function is called with these structure type and structure values as arguments. If the result of the function call involves structure values, these values are converted back to values of the orginal type argument. Consequently, a type-indexed function can be made generic by providing only the cases for Unit, Prod, Sum, and some primitive types.

Thus, a generic version of the *encode* function can be defined as

```
encode ⟨a :: ⋆⟩                :: (encode ⟨a⟩) ⇒ a → Bin
encode ⟨Unit⟩ Unit             = []
encode ⟨Prod α β⟩ (a × b)      = encode ⟨α⟩ a ++ encode ⟨β⟩ b
encode ⟨Sum α β⟩ (Inl a)       = 0 : encode ⟨α⟩ a
encode ⟨Sum α β⟩ (Inr b)       = 1 : encode ⟨β⟩ b
encode ⟨Integer⟩ n             = encodeInteger n
encode ⟨Char⟩ c                = encodeChar c
encode ⟨Float⟩ e               = encodeFloat e.
```

For nullary products, i.e., fieldless data constructors, no bits need to be emitted. Binary products, which represent consecutive fields of a constructor, are encoded by concatenating the recursive encodings of their components. For each choice between two data constructors, represented by a binary sum, a single bit is produced. For the primitive types Integer, Char, and Float, we assume we have handwritten, primitive encoders *encodeInteger*, *encodeChar*, and *encodeFloat* available.

The corresponding generic version of *decodes* is given by

```
decodes ⟨a :: ⋆⟩               :: (decodes ⟨a⟩) ⇒ Bin → (a, Bin)
decodes ⟨Unit⟩ bin             = (Unit, bin)
decodes ⟨Prod α β⟩ bin         = let (a, bin')  = decodes ⟨α⟩ bin
                                     (b, bin'') = decodes ⟨β⟩ bin'
                                 in  (a × b, bin'')
decodes ⟨Sum α β⟩ []           = error "decodes"
```

$$
\begin{array}{ll}
\mathit{decodes}\ \langle \text{Sum}\ \alpha\ \beta \rangle\ (\mathtt{0} : \mathit{bin}) = & \textbf{let}\ (a, \mathit{bin}') = \mathit{decodes}\ \langle \alpha \rangle\ \mathit{bin} \\
& \textbf{in}\ \ (\mathit{Inl}\ a, \mathit{bin}') \\
\mathit{decodes}\ \langle \text{Sum}\ \alpha\ \beta \rangle\ (\mathtt{1} : \mathit{bin}) = & \textbf{let}\ (b, \mathit{bin}') = \mathit{decodes}\ \langle \beta \rangle\ \mathit{bin} \\
& \textbf{in}\ \ (\mathit{Inr}\ b, \mathit{bin}') \\
\mathit{decodes}\ \langle \text{Integer} \rangle\ \mathit{bin} & = \mathit{decodesInteger}\ \mathit{bin} \\
\mathit{decodes}\ \langle \text{Char} \rangle\ \mathit{bin} & = \mathit{decodesChar}\ \mathit{bin} \\
\mathit{decodes}\ \langle \text{Float} \rangle\ \mathit{bin} & = \mathit{decodesFloat}\ \mathit{bin}
\end{array}
$$

Again, we assume the availability of primitive decoders for the types Integer, Char, and Float.

Generic Application

Generic Haskell automatically takes care of the required conversions between 'normal' types and values and their structural representations.

For instance, if we apply the generic encoder on the type Fork Bool, as in

$$\mathit{encode}\ \langle \text{Fork Bool} \rangle\ (\mathit{Fork}\ \mathit{False}\ \mathit{True}),$$

the arguments are converted to their structural representations:

$$\mathit{encode}\ \langle \text{Prod Bool Bool} \rangle\ (\mathit{False} \times \mathit{True}).$$

Next, the case for binary products gives

$$\mathit{encode}\ \langle \text{Bool} \rangle\ \mathit{False} +\!\!+ \mathit{encode}\ \langle \text{Bool} \rangle\ \mathit{True}.$$

To proceed, the Boolean arguments are converted to structural representations,

$$\begin{array}{l}
\mathit{encode}\ \langle \text{Sum Unit Unit} \rangle\ (\mathit{Inl}\ \mathit{Unit}) \\
\quad +\!\!+ \mathit{encode}\ \langle \text{Sum Unit Unit} \rangle\ (\mathit{Inr}\ \mathit{Unit}),
\end{array}$$

after which the cases for binary sums apply:

$$\mathtt{0} : \mathit{encode}\ \langle \text{Unit} \rangle\ \mathit{Unit} +\!\!+ \mathtt{1} : \mathit{encode}\ \langle \text{Unit} \rangle\ \mathit{Unit}.$$

Finally, the case for nullary products yields

$$[\mathtt{0}, \mathtt{1}].$$

Decoding is done in a similar fashion:

$$\mathit{decodes}\ \langle \text{Fork Bool} \rangle\ [0, 1]$$

is converted to

$$\mathit{decodes}\ \langle \text{Prod Bool Bool} \rangle\ [0, 1],$$

which reduces to

$$\begin{array}{l}
\textbf{let}\ (a, \mathit{bin}') \ = \mathit{decodes}\ \langle \text{Bool} \rangle\ [0, 1] \\
\quad\ (b, \mathit{bin}'') = \mathit{decodes}\ \langle \text{Bool} \rangle\ \mathit{bin}' \\
\textbf{in}\ \ (a \times b, \mathit{bin}'').
\end{array}$$

By converting the type Bool to the structure type Sum Unit Unit, we obtain

$$\begin{array}{l}\textbf{let } (a, bin') \;= \mathit{decodes}\ \langle \text{Sum Unit Unit}\rangle\ [0,1] \\ \quad\ (b, bin'') = \mathit{decodes}\ \langle \text{Sum Unit Unit}\rangle\ bin' \\ \textbf{in }\ (a \times b, bin'')\end{array}$$

and then the cases for *decodes* on Sum and Unit, omitting a few steps, give

$$(\mathit{Inl\ Unit} \times \mathit{Inr\ Unit}, [\,]).$$

It remains to convert the first component of this pair to an element of Fork Bool:

$$(\mathit{Fork\ False\ True}, [\,]).$$

4.3 Extensions

In this section, we discuss some extensions to the machinery of Section 4.2 for defining and applying generic functions. In particular, we introduce local redefinitions, generic abstractions, generic functions with non-generic type parameters, default cases, and generic functions with multiple type parameters.

Local Redefinition

Consider the generic version of equality:

$$\begin{array}{ll}\mathit{eq}\ \langle \text{a} :: \star\rangle & :: (\mathit{eq}\ \langle \text{a}\rangle) \Rightarrow \text{a} \to \text{a} \to \text{Bool} \\ \mathit{eq}\ \langle \text{Unit}\rangle\ \mathit{Unit\ Unit} & = \mathit{True} \\ \mathit{eq}\ \langle \text{Prod}\ \alpha\ \beta\rangle\ (a_1 \times b_1)\ (a_2 \times b_2) & = \mathit{eq}\ \langle\alpha\rangle\ a_1\ a_2 \wedge \mathit{eq}\ \langle\beta\rangle\ b_1\ b_2 \\ \mathit{eq}\ \langle \text{Sum}\ \alpha\ \beta\rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{eq}\ \langle\alpha\rangle\ a_1\ a_2 \\ \mathit{eq}\ \langle \text{Sum}\ \alpha\ \beta\rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inr}\ b_2) & = \mathit{False} \\ \mathit{eq}\ \langle \text{Sum}\ \alpha\ \beta\rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inl}\ a_2) & = \mathit{False} \\ \mathit{eq}\ \langle \text{Sum}\ \alpha\ \beta\rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{eq}\ \langle\beta\rangle\ b_1\ b_2 \\ \mathit{eq}\ \langle \text{Integer}\rangle\ n_1\ n_2 & = \mathit{eqInt}\ n_1\ n_2 \\ \mathit{eq}\ \langle \text{Char}\rangle\ c_1\ c_2 & = \mathit{eqChar}\ c_1\ c_2 \\ \mathit{eq}\ \langle \text{Float}\rangle\ e_1\ e_2 & = \mathit{eqFloat}\ e_1\ e_2.\end{array}$$

Assuming that *eqChar* only produces *True* if its argument characters are indeed equal and *False* otherwise, the call

$$\mathit{eq}\ \langle[\text{Char}]\rangle\ [\texttt{'S'},\texttt{'o'},\texttt{'l'}]\ [\texttt{'s'},\texttt{'o'},\texttt{'l'}]$$

reduces to *False*, since `'S'` and `'s'` denote different characters.

Local redefinition [59] allows us to locally replace some of the functionality of a generic function by a custom implementation. For instance, the local redefinition in the expression

$$\begin{array}{l}\textbf{let } \mathit{eq}\ \langle\alpha\rangle\ c_1\ c_2 = \mathit{eqChar}\ (\mathit{toUpper}\ c_1)\ (\mathit{toUpper}\ c_2) \\ \textbf{in }\ \mathit{eq}\ \langle[\alpha]\rangle\ [\texttt{'S'},\texttt{'o'},\texttt{'l'}]\ [\texttt{'s'},\texttt{'o'},\texttt{'l'}]\end{array}$$

causes the character lists $[\texttt{'S'},\texttt{'o'},\texttt{'l'}]$ and $[\texttt{'s'},\texttt{'o'},\texttt{'l'}]$ to be tested for equality in a case-insensitive way: it makes the expression evaluate to *True*.

Generic Abstractions

Generic abstractions [19] allow common idioms that make use of generic functions to be captured as new generic functions.

For example, to capture case-insensitive equality checks for data structures containing characters, we define the generic abstraction *eqCI*,

$$
\begin{array}{l}
\mathit{eqCI}\ \langle \mathrm{f} :: \star \to \star \rangle \qquad :: (\mathit{eq}\ \langle \mathrm{f} \rangle) \Rightarrow \mathrm{f\ Char} \to \mathrm{f\ Char} \to \mathrm{Bool} \\
\mathit{eqCI}\ \langle \varphi :: \star \to \star \rangle\ \mathit{cs}_1\ \mathit{cs}_2 = \\
\quad \textbf{let}\ \mathit{eq}\ \langle \alpha \rangle\ c_1\ c_2 = \mathit{eqChar}\ (\mathit{toUpper}\ c_1)\ (\mathit{toUpper}\ c_2) \\
\quad \textbf{in}\ \ \mathit{eq}\ \langle \varphi\ \alpha \rangle\ \mathit{cs}_1\ \mathit{cs}_2.
\end{array}
$$

Generic abstractions are syntactically distinguished from local redefinitions by explicit kind annotations in the type indexes of their definitions.

Non-generic Type Parameters

The generic function *collect* collects values from a data structure:

$$
\begin{array}{ll}
\mathit{collect}\ \langle \mathrm{a} :: \star \mid \mathrm{c} :: \star \rangle & :: (\mathit{collect}\ \langle \mathrm{a} \mid \mathrm{c} \rangle) \Rightarrow \mathrm{a} \to [\mathrm{c}] \\
\mathit{collect}\ \langle \mathrm{Unit} \rangle\ \mathit{Unit} & = [\,] \\
\mathit{collect}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (a \times b) & = \mathit{collect}\ \langle \alpha \rangle\ a +\!\!+ \mathit{collect}\ \langle \beta \rangle\ b \\
\mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inl}\ a) & = \mathit{collect}\ \langle \alpha \rangle\ a \\
\mathit{collect}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inr}\ b) & = \mathit{collect}\ \langle \beta \rangle\ b \\
\mathit{collect}\ \langle \mathrm{Integer} \rangle\ n & = [\,] \\
\mathit{collect}\ \langle \mathrm{Char} \rangle\ c & = [\,] \\
\mathit{collect}\ \langle \mathrm{Float} \rangle\ e & = [\,].
\end{array}
$$

The type parameter c in the type signature of *collect* is an example of a so-called *non-generic type parameter*. It indicates that *collect* is parametrically polymorphic in the type variable c. The parameter c is passed unchanged to the recursive calls of *collect*; it is global to the definiton of the generic function. Non-generic type parameters only appear in type signatures, where they are placed to the right of vertical bars in type indexes.

Note that, on itself, the function *collect* does not seem very useful: for finite and fully defined values it always returns the empty list. However, in combination with local redefinition it can be used to compute, for example, the breadth-first traversal of a perfect tree:

$$
\begin{array}{l}
\mathit{bftP} \quad :: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Perfect\ a} \to [\mathrm{a}] \\
\mathit{bftP}\ \mathit{as} = \textbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ a = a \\
\qquad\qquad \textbf{in}\ \ \mathit{collect}\ \langle \mathrm{Perfect}\ \alpha \rangle\ \mathit{as}.
\end{array}
$$

This idiom can be captured in a more general form by means of a generic abstraction:

$$
\begin{array}{ll}
\mathit{flatten}\ \langle \mathrm{f} :: \star \to \star \mid \mathrm{a} :: \star \rangle & :: (\mathit{collect}\ \langle \mathrm{f} \mid \mathrm{a} \rangle) \Rightarrow \mathrm{f\ a} \to [\mathrm{a}] \\
\mathit{flatten}\ \langle \varphi :: \star \to \star \rangle\ \mathit{as} & = \textbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ a = [a] \\
& \quad \textbf{in}\ \ \mathit{collect}\ \langle \varphi\ \alpha \rangle\ \mathit{as}.
\end{array}
$$

Actually, the generic function *collect* itself can be generalized to the notion of a generic reduction or crush [63]. (See, for example, Hinze [33].) Reductions are constituted by monoids; the *collect* reduction arises from the concatenation monoid on lists (cf. Section 3.2).

Default Cases

Consider the data type Term of lambda terms,

$$\begin{array}{l}\textbf{data}\ \text{Variable} = \mathit{Variable}\ \text{String}\\ \textbf{data}\ \text{Term} \quad = \mathit{Var}\ \text{Variable}\\ \qquad\qquad\ \mid \mathit{Abs}\ \text{Variable Term}\\ \qquad\qquad\ \mid \mathit{App}\ \text{Term Term},\end{array}$$

and the function *varcollect* that collects all variables that appear in a given lambda term,

$$\begin{array}{ll}\mathit{varcollect} & :: \text{Term} \to [\text{Variable}]\\ \mathit{varcollect}\ (\mathit{Var}\ x) & = [x]\\ \mathit{varcollect}\ (\mathit{Abs}\ x\ t) & = x : \mathit{varcollect}\ t\\ \mathit{varcollect}\ (\mathit{App}\ t_1\ t_2) & = \mathit{varcollect}\ t_1 +\!\!+ \mathit{varcollect}\ t_2.\end{array}$$

A stand-alone variable is put in a singleton list. The variables appearing in a lambda abstraction are its formal parameter and the variables that appear in its body. For applications, we recursively collect the variables of the function and the argument, and concatenate the results.

If we enrich the term language—for instance, by adding constructs like declarations, a fixed-point operator, syntactic sugar for numerals, pairs, booleans, etc.—the data type Term needs to be extended with additional data constructors and, accordingly, the definition of the function *varcollect* needs to be extended with cases for the new constructors. Ideally, we could just rewrite *varcollect* in terms of the generic function *collect*, in such a way that its definition does not need to be adapted each time a change is made to the declaration of the type Term. Unfortunately, *varcollect* cannot be captured as a local redefiniton of *collect*, unless we parameterize Term with the type of variables. In general, it is not acceptable to adjust the definition of a data type, just to accomodate the implementation of some function on its elements.

Default cases [19] allow us to derive a generic function that is an exact copy of another generic function, except for some explicit adjustments. For instance, using default cases, the function *varcollect* can be derived from *collect* as follows:

$$\begin{array}{l}\mathit{varcollect}\ \langle \text{a} :: \star\rangle \quad :: (\mathit{varcollect}\ \langle \text{a}\rangle) \Rightarrow \text{a} \to [\text{Variable}]\\ \mathit{varcollect}\ \textbf{extends}\ \mathit{collect}\\ \mathit{varcollect}\ \langle \text{Variable}\rangle\ x = [x].\end{array}$$

The keyword **extends** indicates that the default cases for *varcollect* are provided by *collect*—i.e., whenever *varcollect* is called with a type argument that cannot be matched against the patterns in its signature, an appropriate case is derived from *collect*. More specifically, a Generic Haskell implementation treats the function *varcollect* as if it were given by

$$\begin{array}{ll}\mathit{varcollect}\ \langle \text{a} :: \star\rangle & :: (\mathit{varcollect}\ \langle \text{a}\rangle) \Rightarrow \text{a} \to [\text{Variable}]\\ \mathit{varcollect}\ \langle \text{Variable}\rangle\ x & = [x]\\ \mathit{varcollect}\ \langle \text{Unit}\rangle\ \mathit{Unit} & = [\,]\\ \mathit{varcollect}\ \langle \text{Prod}\ \alpha\ \beta\rangle\ (a \times b) & = \mathit{varcollect}\ \langle\alpha\rangle\ a +\!\!+ \mathit{varcollect}\ \langle\beta\rangle\ b\\ \mathit{varcollect}\ \langle \text{Sum}\ \alpha\ \beta\rangle\ (\mathit{Inl}\ a) & = \mathit{varcollect}\ \langle\alpha\rangle\ a\end{array}$$

$$
\begin{array}{ll}
\mathit{varcollect}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inr}\ b) & = \mathit{varcollect}\ \langle \beta \rangle\ b \\
\mathit{varcollect}\ \langle \mathrm{Integer} \rangle\ n & = [\,] \\
\mathit{varcollect}\ \langle \mathrm{Char} \rangle\ c & = [\,] \\
\mathit{varcollect}\ \langle \mathrm{Float} \rangle\ e & = [\,].
\end{array}
$$

Note that *varcollect* now not just works for Term, but for all data types. Actually, since there is no explicit case for Term, *varcollect* processes Term values according to their structural representations, i.e. as values of the structure type

Sum Variable (Sum (Prod Variable Term) (Prod Term Term)).

Multiple Type Parameters

Mapping functions lift functions over the element types of data structures to functions over data structures. The definitions of mapping functions follow a very rigid scheme, making them excellent candidates for generic implementation. For instance, the mapping functions for lists and external binary search trees are given by

$$
\begin{array}{ll}
\mathit{mapList} & :: \forall(\mathrm{a} :: \star)\ (\mathrm{b} :: \star)\ .\ (\mathrm{a} \to \mathrm{b}) \to [\mathrm{a}] \to [\mathrm{b}] \\
\mathit{mapList}\ f\ [\,] & = [\,] \\
\mathit{mapList}\ f\ (a : as) & = f\ a : \mathit{mapList}\ f\ as
\end{array}
$$

and

$$
\begin{array}{ll}
\mathit{mapTree} & :: \\
\quad \forall(\mathrm{a} :: \star)\ (\mathrm{b} :: \star)\ (\mathrm{c} :: \star)\ (\mathrm{d} :: \star)\ . & \\
\qquad (\mathrm{a} \to \mathrm{c}) \to (\mathrm{b} \to \mathrm{d}) \to \mathrm{Tree\ a\ b} \to \mathrm{Tree\ c\ d} & \\
\mathit{mapTree}\ f\ g\ (\mathit{Tip}\ a) & = \mathit{Tip}\ (f\ a) \\
\mathit{mapTree}\ f\ g\ (\mathit{Node}\ l\ b\ r) & = \mathit{Node}\ (\mathit{mapTree}\ f\ g\ l)\ (g\ b)\ (\mathit{mapTree}\ f\ g\ r).
\end{array}
$$

The function *mapList* takes two arguments: a function and a list. It computes a new list by applying the argument function to all elements of the argument list. Likewise, the function *mapTree* takes three arguments: two functions and a tree. A new tree is constructed by applying the first function to all leafs of the argument tree and applying the second function to all labels. For example, the call

$$\mathit{mapList}\ \mathit{chr}\ [115, 111, 108]$$

yields `['s','o','l']`, while the call

$$\mathit{mapTree}\ \mathit{ord}\ \mathit{id}\ (\mathit{Node}\ (\mathit{Tip}\ \texttt{'s'})\ 111\ (\mathit{Tip}\ \texttt{'l'}))$$

reduces to $\mathit{Node}\ (\mathit{Tip}\ 115)\ 111\ (\mathit{Tip}\ 108)$.

In Generic Haskell, the generic mapping function is defined as

$$
\begin{array}{ll}
\mathit{map}\ \langle \mathrm{a} :: \star, \mathrm{b} :: \star \rangle & :: (\mathit{map}\ \langle \mathrm{a}, \mathrm{b} \rangle) \Rightarrow \mathrm{a} \to \mathrm{b} \\
\mathit{map}\ \langle \mathrm{Unit} \rangle\ \mathit{Unit} & = \mathit{Unit} \\
\mathit{map}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (a \times b) & = \mathit{map}\ \langle \alpha \rangle\ a \times \mathit{map}\ \langle \beta \rangle\ b \\
\mathit{map}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inl}\ a) & = \mathit{Inl}\ (\mathit{map}\ \langle \alpha \rangle\ a) \\
\mathit{map}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inr}\ b) & = \mathit{Inr}\ (\mathit{map}\ \langle \beta \rangle\ b)
\end{array}
$$

$$
\begin{aligned}
&\mathit{map}\ \langle \text{Integer} \rangle\ n &&= n \\
&\mathit{map}\ \langle \text{Char} \rangle\ c &&= c \\
&\mathit{map}\ \langle \text{Float} \rangle\ e &&= e.
\end{aligned}
$$

The interesting part of this declaration is the type signature for *map*, since the type index contains two type variables. By default, a call to a type-indexed function that has multiple type parameters causes all parameters to be instantiated with the same type—i.e, the type argument. However, local redefinition allows us to instantiate the parameters with different types. For example, in the call

$$\mathit{map}\ \langle [\text{Integer}] \rangle\ [115, 111, 108]$$

both type parametes of *map* are instantiated with the type $[\text{Integer}]$, i.e.,

$$\mathit{map}\ \langle [\text{Integer}] \rangle :: [\text{Integer}] \to [\text{Integer}].$$

Yet, in the expression

$$
\begin{aligned}
&\textbf{let}\ \mathit{map}\ \langle \alpha \rangle\ n = \mathit{chr}\ n \\
&\textbf{in}\ \ \mathit{map}\ \langle [\alpha] \rangle\ [115, 111, 108]
\end{aligned}
$$

the local redefiniton on α instantiates the type parameters of *map* with Integer and Char, and therefore, within the scope of the redefinition, we have

$$\mathit{map}\ \langle [\alpha] \rangle :: [\text{Integer}] \to [\text{Char}].$$

In the first example the call to map yields $[115, 111, 108]$, while in the second it results in `['s','o','l']`. So, *map* basicly behaves like a generic identity function[3], permitting mapping behaviour to be derived through local redefinition.

4.4 Notes

The translation from a source language with type-indexed functions and structural polymorphism to a functional target language is called specialization and is discussed thoroughly in the literature on Generic Haskell [35, 92, 39, 58].

The term 'generic abstraction' was coined by Clarke and Löh [19] in their paper 'Generic Haskell, specifically', but the underlying idea was already used before—albeit informally—by Hinze [32]. In the same paper, Clarke and Löh introduced so-called copy lines, which later evolved in default cases. Both generic abstraction and default cases were adapted by Löh [58] to fit in the framework of Dependency-style Generic Haskell [59].

[3] In fact, *map* and the generic identity function *copy* only differ in their types:

$$
\begin{aligned}
&\mathit{map}\ \langle \mathsf{a} :: \star, \mathsf{b} :: \star \rangle :: (\mathit{map}\ \langle \mathsf{a}, \mathsf{b} \rangle) \Rightarrow \mathsf{a} \to \mathsf{b} \\
&\mathit{copy}\ \langle \mathsf{a} :: \star \rangle\ \ \ :: (\mathit{copy}\ \langle \mathsf{a} \rangle) \Rightarrow \mathsf{a} \to \mathsf{a}.
\end{aligned}
$$

Except for those, *map* and *copy* have identical definitions. It follows that type-indexed functions do not have principal types.

Part II

Views

Chapter 5

Generic Views on Data Types

In this chapter, we discuss the concept of views on data types (Section 5.1). A view allows values of a single data type to be matched against an additional set of constructors. In Section 5.2, we argue that structural representations in Generic Haskell essentially extend this idea to whole classes of data types. For such an extended view on data types, we introduce the term 'generic view'. A formal treatment is given in Section 5.3.

5.1 Views

Abstraction and induction are important concepts to functional programmers.

Data abstraction or representation hiding helps reducing data types to their essentials. By specifying a data type only in terms of its interface, i.e., the operations it supports, the implementor of the type is free to change the actual representation of the data type at any time. She can even decide to use two or more representations simultaneously. Decoupling the definition of a data type from its usage facilitates modularity and improves the maintainability of programs. In Haskell, data abstraction is achieved by hiding the concrete implementation of a data type behind a module boundary. The module then only exports the relevant type constructor and the functions that make up the interface of the abstract data type.

Induction is one of the mathematician's most powerful tools. In functional programming, induction plays an important rôle in proving properties of programs as well as in defining functions on algebraic data types. Inductive function definitions in Haskell are facilitated by pattern matching.

Unfortunately, abstract data types and pattern matching do not go along too well: an abstract type effectively hides its data constructors, whereas the visibitity of those is essential to pattern matching. Hence, often the implementor of a data type is confronted with a trade-off: should she enforce modularity by making the type abstract or should she facilitate inductive definitions by exposing the type's implementation?

This section centers around a proposal of Wadler [89], arguing that the trade-off between abstraction and induction can be avoided by extending functional languages with the notion of *views*. A view allows abstract data types to be perceived as concrete types, so they can be subjected to pattern matching.

Here, we illustrate the concept of views by considering two representations of the natural numbers.

Inductive Definitions

A straightforward representation of the natural numbers, $0..$, based on the Peano axioms, is

$$\textbf{data}\ \text{Peano} = \mathit{Zero} \mid \mathit{Succ}\ \text{Peano}.$$

This definition expresses that zero is a natural number; and that, if n is a number, then the successor of n is also a number.

The definition of Peano facilitates inductive definitions of functions on natural numbers. For instance, addition, multiplication, and exponentiation of Peano naturals are defined by

$$\begin{array}{l}
\textbf{infixl}\ 6\ \oplus \\
\textbf{infixl}\ 7\ \circledast \\
\textbf{infixr}\ 8\ \wedge
\end{array}$$

$$\begin{array}{ll}
(\oplus) & :: \text{Peano} \to \text{Peano} \to \text{Peano} \\
m \oplus \mathit{Zero} & = m \\
m \oplus \mathit{Succ}\ n & = \mathit{Succ}\ (m \oplus n)
\end{array}$$

$$\begin{array}{ll}
(\circledast) & :: \text{Peano} \to \text{Peano} \to \text{Peano} \\
m \circledast \mathit{Zero} & = \mathit{Zero} \\
m \circledast \mathit{Succ}\ n & = m \oplus m \circledast n
\end{array}$$

$$\begin{array}{ll}
(\wedge) & :: \text{Peano} \to \text{Peano} \to \text{Peano} \\
m \wedge \mathit{Zero} & = \mathit{Succ}\ \mathit{Zero} \\
m \wedge \mathit{Succ}\ n & = m \circledast m \wedge n.
\end{array}$$

Besides their elegance, inductive definitions like these have several favorable properties. For example, they very well accomodate proofs by structural induction [13].

Abstract Data Types

Despite its advantages, the Peano representation of natural numbers is a very ineffcient one. Large numbers are represented by a large number of constructor applications, consuming large amounts of space and slowing down the execution of programs that operate on them.

For integers subsume natural numbers, the built-in Integer type makes a far more efficient representation of naturals:

$$\textbf{data}\ \text{Nat} = \mathit{Nat}\ \text{Integer}.$$

Still, this representation suffers from an obvious drawback: the type Nat has too many elements. For instance, $\mathit{Nat}\ (-2)$ denotes a valid value of Nat; yet, (-2) is not a natural number.

To overcome this problem, one typically makes Nat an *abstract data type*—i.e., the type declaration of Nat is hidden inside a module that only exports the type constructor and a handful of auxiliary functions that operate on values of

Nat. As a consequence, the data constructor *Nat* is not visible from anywhere outside the module. Within the module, the invariant is to be maintained that the Integer component of a Nat value is never negative. For the construction of Nat values from outside the module, a special auxiliary function is exported:

```
mkNat              :: Integer → Nat
mkNat n | n < 0  = error "mkNat"
        | n ⩾ 0  = Nat n.
```

Invariant-maintaining constructor functions like *mkNat* are often called *smart constructors*.

An unfortunate consequence of making a data type abstract is that it cripples pattern matching. Hiding the data constructor *Nat* makes that, outside the defining module, Nat values can only be matched against variable patterns. As a result, function definitions become less elegant and proofs gain in complexity.

View Constructors

The question we ask ourselves is: Can we get the best of both worlds? Can we have, hidden behind a module boundary, the efficient but less accurate definition of Nat, without sacrificing the power and elegance of pattern matching against constructors like *Zero* and *Succ*?

To answer this question in the affirmative, we require a relatively simple language extension: views. A view on a data type basically is a set of additional data constructors against which values of the type can be matched. These additional constructors are commonly called *view constructors*. Yet, they exclusively act as destructors, for their appearance is restricted to the left-hand sides of function definitions.

For instance, a view that allows Nat values to be matched against view constructors *Zero* and *Succ* is given by

```
view Peano of Nat = Zero | Succ Nat
  where peano (Nat n) | n == 0 = Zero
                      | n > 0  = Succ (mkNat (n − 1)).
```

Note that the view is non-recursive: the view constructor *Succ* takes an argument of type Nat.

The special function *peano* is called a *view transformation*: it specifies how values of Nat are matched against *Zero* and *Succ*. A value *Nat n* is successfully matched against *Zero*, if *n* evaluates to 0. Likewise, *Nat n* is successfully matched against *Succ m*, if *n* evaluates to a positive value, and *Nat* $(n-1)$ and *m* match. View transformations are the only functions that can use view constructors at the right-hand sides of their definitions.

Using this view, addition, multiplication, and exponentiation of Nat values can be defined as follows:

```
infixl 6 ⊕
infixl 7 ⊛
infixr 8 ↑

(⊕)            :: Nat → Nat → Nat
```

$$
\begin{array}{ll}
m \oplus \mathit{Zero} & = m \\
m \oplus \mathit{Succ}\ n & = m \oplus n \oplus \mathit{mkNat}\ 1
\end{array}
$$

$$
\begin{array}{ll}
(\circledast) & :: \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \\
m \circledast \mathit{Zero} & = \mathit{mkNat}\ 0 \\
m \circledast \mathit{Succ}\ n & = m \oplus m \circledast n
\end{array}
$$

$$
\begin{array}{ll}
(\text{人}) & :: \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \\
m \text{人} \mathit{Zero} & = \mathit{mkNat}\ 1 \\
m \text{人} \mathit{Succ}\ n & = m \circledast m \text{人} n.
\end{array}
$$

Notice that the view constructors *Zero* and *Succ* only appear in patterns at the left-hand sides of the definitions. At the right-hand sides, the smart constructor *mkNat* is used to construct new values of Nat.

Multiple Views

A data type can expose any number of views. For example, consider the view EvenOrOdd of Nat:

$$
\begin{array}{l}
\textbf{view}\ \mathrm{EvenOrOdd}\ \textbf{of}\ \mathrm{Nat} = \mathit{ZeroE} \mid \mathit{Even}\ \mathrm{Nat} \mid \mathit{Odd}\ \mathrm{Nat} \\
\quad \textbf{where}\ \mathit{evenOrOdd}\ (\mathit{Nat}\ n) \\
\qquad\qquad \mid n == 0 \qquad\qquad\qquad\quad = \mathit{ZeroE} \\
\qquad\qquad \mid n > 0 \wedge n\ \text{`}\mathit{mod}\text{`}\ 2 == 0 = \mathit{Even}\ (\mathit{mkNat}\ (n\ \text{`}\mathit{div}\text{`}\ 2)) \\
\qquad\qquad \mid n > 0 \wedge n\ \text{`}\mathit{mod}\text{`}\ 2 == 1 = \mathit{Odd}\ (\mathit{mkNat}\ ((n-1)\ \text{`}\mathit{div}\text{`}\ 2)).
\end{array}
$$

In this view, the value $\mathit{Nat}\ 0$ is viewed as *ZeroE*. A value $\mathit{Nat}\ n$, for even n, is viewed as $\mathit{Even}\ (\mathit{Nat}\ (n\ \text{`}\mathit{div}\text{`}\ 2))$. A value $\mathit{Nat}\ n$, for odd n, is viewed as $\mathit{Odd}\ (\mathit{Nat}\ ((n-1)\ \text{`}\mathit{div}\text{`}\ 2))$. So, for example, $\mathit{Nat}\ 2$ is viewed as $\mathit{Even}\ (\mathit{Nat}\ 1)$ and $\mathit{Nat}\ 3$ as $\mathit{Odd}\ (\mathit{Nat}\ 1)$. Again, the view is non-recursive.

Using the EvenOrOdd view in conjunction with the Peano view, a more efficient version of exponentiation of Nat values can be defined:

$$
\begin{array}{l}
\textbf{infixr}\ 8\ \text{丛} \\
\\
\begin{array}{ll}
(\text{丛}) & :: \mathrm{Nat} \to \mathrm{Nat} \to \mathrm{Nat} \\
m \text{丛} \mathit{ZeroE} & = \mathit{mkNat}\ 1 \\
m \text{丛} \mathit{Even}\ n & = (m \circledast m) \text{丛} n \\
m \text{丛} \mathit{Odd}\ n & = m \circledast (m \circledast m) \text{丛} n.
\end{array}
\end{array}
$$

This definition captures the well-known divide-and-conquer approach to exponentiation.

5.2 Towards Generic Views

A view allows a programmer to perceive values of a certain data type as if they were defined in terms of a set of alternative constructors. In Generic Haskell, structural polymorphism allows a programmer to perceive values of a whole class of data types as if they were defined in terms of the constructors *Unit*, $(\times)$, *Inl*, and *Inr*. Hence, one can think of structural polymorphism as a mechanism that extends the concept of views from alternative representations of single data types to alternative representations of classes of data types. We say that Generic Haskell provides a *generic view* on data types.

Alternative Views on Data Types

As illustrated in the previous section, a data type can expose multiple non-generic views. In contrast, Generic Haskell provides only one generic view on data types: the view that allows types and values to be perceived in terms of their structural representations as nested sums and products.

The main contribution of this thesis is that we consider the possibility of having multiple generic views on data types. The intuition behind this is that, analogously to the usage of non-generic views, some generic functions behave more efficient when defined in terms of another generic view. Moreover, in some cases, generic functions that are hard or even impossible to define in one view, can be elegantly defined in another view.

In the sequel, the 'classic' generic view in terms of nested sums and products is referred to as the *standard view* on types. In later chapters, several alternative views are discussed. Here, we focus on what exactly constitutes a generic view.

Conversions

A generic view maps types and values to structural representations. Structural representations are type and value expressions defined in terms of a, usually, small set of data types. These data types are called *view types*. For example, the view types of the standard view are Unit, Prod, and Sum.

The mapping from types to structure types is partial. For instance, the standard view does not contain a mapping for primitive types like Integer, Char, Float, and $(\rightarrow)$. The set of types for which a view does define a mapping, is called the *view domain* of the view. A generic view provides a kind-preserving algorithm that produces a structural representation for each type in its view domain.

If a generic function is called with a type that does not appear in its signature, the call is lifted to a call for the appropriate structure type. Values of the original type that appear in the arguments of the function call may then need to be matched against constructors of the view types. This requires a view transformation that takes a value of the orginal type to a value of its structure type. For instance, for the standard view, a transformation from values of type Bool,

$$\textbf{data}\ \text{Bool} = \mathit{False} \mid \mathit{True},$$

to values of the structure type Bool°,

$$\textbf{type}\ \text{Bool}^\circ = \text{Sum Unit Unit},$$

is given by the function *fromBool*,

$$\begin{aligned} &\mathit{fromBool} &&:: \text{Bool} \rightarrow \text{Bool}^\circ \\ &\mathit{fromBool\ False} &&= \mathit{Inl\ Unit} \\ &\mathit{fromBool\ True} &&= \mathit{Inr\ Unit}. \end{aligned}$$

For each type in its view domain, a generic view defines a transformation from values to structure values.

Recall that the view constructors of the non-generic views discussed in Section 5.1 were restricted to occur only in patterns at the left-hand sides of function definitions. In contrast, values of the structure types defined by a generic view can also be constructed at the right-hand sides of definitions. Therefore, in addition to transformations from values to structure values, a view should also provide conversions in the opposite direction. Thus, for Booleans, the standard view provides a conversion that behaves like

$$\begin{aligned} &\mathit{toBool} && :: \mathrm{Bool}^\circ \to \mathrm{Bool} \\ &\mathit{toBool}\ (\mathit{Inl}\ \mathit{Unit}) &&= \mathit{False} \\ &\mathit{toBool}\ (\mathit{Inr}\ \mathit{Unit}) &&= \mathit{True}. \end{aligned}$$

To make the usage of generic views intuitive, it is required that the conversion from structure values to values is the inverse of the conversion from values to structure values. In Haskell, we use the data type Conv to pair conversion functions,

$$\textbf{data}\ \mathrm{Conv}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star) = \mathit{Conv}\{\mathit{from} :: \mathrm{a} \to \mathrm{b}, \mathit{to} :: \mathrm{b} \to \mathrm{a}\}.$$

Hence, a pairing of conversion functions for Booleans in the standard view is given by

$$\begin{aligned} &\mathit{convBool} :: \mathrm{Conv\ Bool\ Bool}^\circ \\ &\mathit{convBool} = \mathit{Conv}\{\mathit{from} = \mathit{fromBool}, \mathit{to} = \mathit{toBool}\}. \end{aligned}$$

5.3 Generic Views, Formally

In this section, we formally define generic views in the setting of the core language presented in Chapter 2.

Parameterized Types

As shown in the previous section, structure types in Haskell are declared by means of the **type** construct. This construct is used to define type synonyms, rather than new algebraic data types. In particular, a **type** declaration does not introduce any new data constructors.

Type synonyms are not supported by our language. Therefore, to be able to describe structure types, we extend the language with *parameterized types*. In the sequel, the metavariable u ranges over parameterized types (see Table 5.1[1]). Figure 5.1 shows the updated syntax of the type language. A parameterized type, which takes the form

$$\{\Lambda a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ t$$

is essentially a nesting of type-level lambda abstractions. Parameterized types cannot appear in the syntax tree of a core-language program; they are only used in view definitions.

The kinding rule for parameterized types is presented in Figure 5.2; it is of the form

$$\mathrm{K} \vdash u :: t.$$

[1] Gray boxes in figures and tables indicate 'deltas' with corresponding versions from Chapter 2.

a	type variable	v	value
d	value declaration	x	variable
e	expression	C	data constructor
i	natural number	D	type declaration
j	natural number	P	program
k	natural number	T	type constructor
ℓ	natural number	κ	kind
m	natural number	φ	value substitution
n	natural number	ψ	type substitution
p	pattern	Γ	type environment
t	type	K	kind environment
u	parameterized type		

Table 5.1: Metavariables, replaces Table 2.1

Type declarations

$$D \quad ::= \quad \mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}$$

algebraic data type

Parameterized types

$$u \quad ::= \quad \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ t$$

type-level abstraction

Types

$$\begin{array}{lcll} t & ::= & a & \text{type variable} \\ & | & T & \text{type constructor} \\ & | & (t_1\ t_2) & \text{type application} \\ & | & \forall a :: \kappa\ .\ t & \text{universal quantification} \end{array}$$

Figure 5.1: Syntax of the type language, replaces Figure 2.2

$$\boxed{\mathrm{K} \vdash u :: \kappa}$$

$$\frac{\mathrm{K}\ \{, a_i :: \kappa_i\}^{i\in 1..\ell} \vdash t :: \kappa}{\mathrm{K} \vdash \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ t :: \{\kappa_i \to\}^{i\in 1..\ell}\ \kappa} \quad \text{(k-par)}$$

Figure 5.2: Kinding for parameterized types

Definitions

Using the notion of parameterized types, we can formalize the observation that a view is constituted by a collection of view types and algorithms for the generation of structure types and conversion functions.

Definition 5.1 (**Generic View**). A *generic view* $\mathcal{V}$ consists of a collection of bindings for view types,

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}; \Gamma,$$

a partial mapping from types to structure types,

$$\mathcal{V}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n},$$

and, for each type in the domain of this mapping, conversions between values and structure values,

$$\mathcal{V}[\![D_0]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}}.$$

□

Notice that we allow the mapping from types to structure types to generate zero or more additional declarations for supporting data types. The types introduced by these declarations can be used for the generation of structure types.

For a view to be useful for generic programming with structural polymorphism, we require it to have three essential properties. First, the mapping from types to structure types should preserve kinds.

Definition 5.2 (**Kind Preservation**). A generic view $\mathcal{V}$ with

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}_{\mathcal{V}}; \Gamma_{\mathcal{V}}$$

is *kind preserving* if for each well-formed declaration D_0 of a type constructor T with kind κ,

$$\mathrm{K} \vdash D_0 \rightsquigarrow T :: \kappa; \Gamma_0,$$

for which a structure type u can be derived,

$$\mathcal{V}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n},$$

it follows that under kind environment K',

$$\mathrm{K}' \equiv \mathrm{K}, \mathrm{K}_{\mathcal{V}}\ \{, \mathrm{K}_i\}^{i \in 1..n},$$

the supporting type declarations are well-formed,

$$\{\mathrm{K}' \vdash D_i \rightsquigarrow \mathrm{K}_i; \Gamma_i\}^{i \in 1..n},$$

and the structure type u has the same kind κ as the orginal type T,

$$\mathrm{K}' \vdash u :: \kappa.$$

□

Second, a pair of conversion functions derived from a type declaration should be well-typed and indeed convert between values of the orginal type and values of the structure type.

Definition 5.3 (**Well-typed Conversion**). A view $\mathcal{V}$ with

$$\mathsf{viewtypes}_{\mathcal{V}} \equiv \mathrm{K}_{\mathcal{V}}; \Gamma_{\mathcal{V}}$$

generates *well-typed conversions* if, for each well-formed declaration D_0 of a type constructor T,

$$\begin{aligned} \mathrm{K} &\vdash D_0 \rightsquigarrow \mathrm{K}_0; \Gamma_0 \\ \mathrm{K}_0 &\equiv T :: \{\kappa_i \to\}^{i \in 1..\ell} \star, \end{aligned}$$

for which a structure type t can be derived,

$$\mathcal{V}[\![D_0]\!]^{\mathrm{str}} \equiv \{\Lambda a_i :: \kappa_i \,.\, \}^{i \in 1..\ell}\ t; \{D_i\}^{i \in 1..n},$$

it follows that the corresponding conversion functions e_{from} and e_{to},

$$\mathcal{V}[\![D_0]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}},$$

take values of the orginal data type T to values of the structure type t and vice versa,

$$\begin{aligned} \mathrm{K}, \mathrm{K}_{\mathcal{V}}, \mathrm{K}_0; \Gamma_{\mathcal{V}}, \Gamma_0 &\vdash e_{\mathrm{from}} :: \{\forall a_i :: \kappa_i \,.\, \}^{i \in 1..\ell}\ T\ \{a_i\}^{i \in 1..\ell} \to t \\ \mathrm{K}, \mathrm{K}_{\mathcal{V}}, \mathrm{K}_0; \Gamma_{\mathcal{V}}, \Gamma_0 &\vdash e_{\mathrm{to}} \quad :: \{\forall a_i :: \kappa_i \,.\, \}^{i \in 1..\ell}\ t \to T\ \{a_i\}^{i \in 1..\ell}. \end{aligned}$$

□

Third, the conversion functions from structure values to values should form the inverses of the corresponding functions in the opposite direction.

Definition 5.4 (**Well-behaved Conversion**). A generic view $\mathcal{V}$ produces *well-behaved conversions* if, for each well-formed declaration D of a type constructor T,

$$\begin{aligned} \mathrm{K} &\vdash D \rightsquigarrow \mathrm{K}'; \Gamma' \\ \mathrm{K}' &\equiv T :: \{\kappa_i \to\}^{i \in 1..\ell} \star, \end{aligned}$$

conversion functions e_{from} and e_{to} are generated,

$$\mathcal{V}[\![D]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}},$$

such that e_{to} is the left inverse of e_{from} with respect to function composition—i.e.,

$$\vdash e_{\mathrm{to}}\ (e_{\mathrm{from}}\ v) \rightarrowtail\!\!\!\!\!\to v$$

for each value v of type T,

$$\mathrm{K}, \mathrm{K}'; \Gamma, \Gamma' \vdash v :: T\ \{t_i\}^{i \in 1..\ell}.$$

□

(Note that, for a well-behaved conversion pair, the function that takes values to structure values is injective; thus, a structure type should have at least as many elements as the corresponding orginal type.)

Only views that possess these three properties are considered valid.

Definition 5.5 (**Validity**). A generic view is *valid* if it is kind preserving and generates well-typed, well-behaved conversions. □

5.4 Notes

Haskell already provides an alternative for pattern matching on integer values by means of so-called $(n + k)$ patterns.

Wadler's original proposal did not put restrictions on the usage of view constructors. More specifically, it allowed view constructors to appear at the right-hand sides of function definitions. To accomodate this, additional transformations from view types to normal types need to be defined. However, as it turns out, it is not trivial to define transformations to and from view types that are each other's inverses. But if these transformations are not inverses, equational reasoning becomes troublesome; similar problems arose with lawful types [86] in Miranda[2]—that also address the issue of pattern matching against values of abstract data types—and resulted in their removal from the language. Therefore, a later proposal [15] for extending the Haskell 98 language with views on data types included the restriction that view constructors can only be used to destruct values.

Views, as a mechanism for pattern matching against abstract data types, are also discussed by Burton et al. [14], Palao-Gostanza et al. [72], and Okasaki [71].

Not only does the standard view on data types generate well-behaved conversions, moreover, these conversions form embedding-projection pairs [23]. Views that generate embedding-projection pairs accomodate equational reasoning and make the behaviour of generic applications more intuitive.

Views have also been proposed in the fields of XML [22] and databases [69, 1]. Generic views as proposed by Souza dos Santos et al. [85] are used to automatically convert between two given views. The concept of a generic view as introduced in this chapter does not seem to have been investigated in this field.

The idea of using different sets of data types for inductive definitions of type-indexed functions is common in the world of dependent types [4, 10]. This corresponds to the idea of having views that work on different subsets of the Haskell data types. However, in the approaches we have seen there is no automatic conversion between syntactically definiable data types as offered by the dependently typed programming language into representations as defined by the view or universe.

[2]Miranda is a trademark of Research Software Limited.

Chapter 6

Sums and Products

In this chapter, we discuss the standard view on data types (Section 6.1) and two slight variations: a view that yields a balanced encoding in sums and products (Section 6.2) and a list-like view (Section 6.3). We mainly focus on the algorithms that generate structure types and conversion functions for these views. Furthermore, we briefly show why a balanced-encoding view and a list-like view might be useful.

6.1 Standard View

In Chapter 4, examples were given of how structural representations are derived for a set of archetypical data types. Essentially, for a given type, each constructor gave rise to a term of a sum, while each constructor field was mapped to a factor of a product. The resulting representations constitute the *standard view* on data types.

Conversions

In Section 5.2, the type Conv,

$$\textbf{data}\ \mathrm{Conv}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star) = \mathit{Conv}\{\mathit{from} :: \mathrm{a} \to \mathrm{b}, \mathit{to} :: \mathrm{b} \to \mathrm{a}\},$$

was used to bundle a pair of conversion functions between the type Bool and its structural representation in the standard view. Here, we present the conversions for the structure types that appear in Chapter 4:

$$\begin{array}{ll}
\textbf{data}\ \mathrm{Bool} & = \mathit{False} \mid \mathit{True} \\
\textbf{type}\ \mathrm{Bool}^{\circ} & = \mathrm{Sum\ Unit\ Unit} \\
\\
\mathit{fromBool} & :: \mathrm{Bool} \to \mathrm{Bool}^{\circ} \\
\mathit{fromBool\ False} & = \mathit{Inl\ Unit} \\
\mathit{fromBool\ True} & = \mathit{Inr\ Unit} \\
\\
\mathit{toBool} & :: \mathrm{Bool}^{\circ} \to \mathrm{Bool} \\
\mathit{toBool}\ (\mathit{Inl\ Unit}) & = \mathit{False} \\
\mathit{toBool}\ (\mathit{Inr\ Unit}) & = \mathit{True}
\end{array}$$

```
convBool                    :: Conv Bool Bool°
convBool                    = Conv{from = fromBool, to = toBool}

data Fork (a :: ⋆)          = Fork a a
type Fork° (a :: ⋆)         = Prod a a

fromFork                    :: ∀(a :: ⋆) . Fork a → Fork° a
fromFork (Fork a1 a2)       = a1 × a2

toFork                      :: ∀(a :: ⋆) . Fork° a → Fork a
toFork (a1 × a2)            = Fork a1 a2

convFork                    :: ∀(a :: ⋆) . Conv (Fork a) (Fork° a)
convFork                    = Conv{from = fromFork, to = toFork}

data [a :: ⋆]               = [] | a : [a]
type [a :: ⋆]°              = Sum Unit (Prod a [a])

fromList                    :: ∀(a :: ⋆) . [a] → [a]°
fromList []                 = Inl Unit
fromList (a : as)           = Inr (a × as)

toList                      :: ∀(a :: ⋆) . [a]° → [a]
toList (Inl Unit)           = []
toList (Inr (a × as))       = a : as

convList                    :: ∀(a :: ⋆) . Conv [a] [a]°
convList                    = Conv{from = fromList, to = toList}

data Rose (a :: ⋆)          = Branch a [Rose a]
type Rose° (a :: ⋆)         = Prod a [Rose a]

fromRose                    :: ∀(a :: ⋆) . Rose a → Rose° a
fromRose (Branch a as)      = a × as

toRose                      :: ∀(a :: ⋆) . Rose° a → Rose a
toRose (a × as)             = Branch a as

convRose                    :: ∀(a :: ⋆) . Conv (Rose a) (Rose° a)
convRose                    = Conv{from = fromRose, to = toRose}

data GRose (f :: ⋆ → ⋆) (a :: ⋆)
                            = GBranch a (f (GRose f a))
type GRose° (f :: ⋆ → ⋆) (a :: ⋆)
                            = Prod a (f (GRose f a))

fromGRose                   ::
  ∀(f :: ⋆ → ⋆) (a :: ⋆) . GRose f a → GRose° f a
fromGRose (GBranch a as)    = a × as

toGRose                     ::
  ∀(f :: ⋆ → ⋆) (a :: ⋆) . GRose° f a → GRose f a
toGRose (a × as)            = GBranch a as

convGRose                   ::
  ∀(f :: ⋆ → ⋆) (a :: ⋆) . Conv (GRose f a) (GRose° f a)
convGRose                   = Conv{from = fromGRose, to = toGRose}
```

```
data Tree (a :: ⋆) (b :: ⋆)   = Tip a | Node (Tree a b) b (Tree a b)
type Tree° (a :: ⋆) (b :: ⋆) =
  Sum a (Prod (Tree a b) (Prod b (Tree a b)))

fromTree                      :: ∀(a :: ⋆) (b :: ⋆) . Tree a b → Tree° a b
fromTree (Tip a)              = Inl a
fromTree (Node l b r)         = Inr (l × (b × r))

toTree                        :: ∀(a :: ⋆) (b :: ⋆) . Tree° a b → Tree a b
toTree (Inl a)                = Tip a
toTree (Inr (l × (b × r)))    = Node l b r

convTree                      =
  ∀(a :: ⋆) (b :: ⋆) . Conv (Tree a b) (Tree° a b)
convTree                      = Conv{from = fromTree, to = toTree}

data Perfect (a :: ⋆)         = ZeroP a | SuccP (Perfect (Fork a))
type Perfect° (a :: ⋆)        = Sum a (Perfect (Fork a))

fromPerfect                   :: ∀(a :: ⋆) . Perfect a → Perfect° a
fromPerfect (ZeroP a)         = Inl a
fromPerfect (SuccP as)        = Inr as

toPerfect                     :: ∀(a :: ⋆) . Perfect° a → Perfect a
toPerfect (Inl a)             = ZeroP a
toPerfect (Inr as)            = SuccP as

convPerfect                   :: ∀(a :: ⋆) . Conv (Perfect a) (Perfect° a)
convPerfect                   = Conv{from = fromPerfect, to = toPerfect}

data Sequ (a :: ⋆)            =
  EndS | ZeroS (Sequ (Fork a)) | OneS a (Sequ (Fork a))
type Sequ° (a :: ⋆)           =
  Sum Unit (Sum (Sequ (Fork a)) (Prod a (Sequ (Fork a))))

fromSequ                      :: ∀(a :: ⋆) . Sequ a → Sequ° a
fromSequ EndS                 = Inl Unit
fromSequ (ZeroS as)           = Inr (Inl as)
fromSequ (OneS a as)          = Inr (Inr (a × as))

toSequ                        :: ∀(a :: ⋆) . Sequ° a → Sequ a
toSequ (Inl Unit)             = EndS
toSequ (Inr (Inl as))         = ZeroS as
toSequ (Inr (Inr (a × as)))   = OneS a as

convSequ                      :: ∀(a :: ⋆) . Conv (Sequ a) (Sequ° a)
convSequ                      = Conv{from = fromSequ, to = toSequ}.
```

The definitions for the conversion functions closely follow the structure of the type declarations. Note that the expressions that appear at the right-hand sides of the 'from' functions reappear as patterns at the left-hand sides of the 'to' functions. Likewise, the patterns at the left-hand side of the 'from' functions are used at the right-hand sides of the 'to' funtions to reconstruct the values of the original data type.

In the remainder of this section, we present the algorithms that generate structure types and conversions for the standard view. These algorithms are

$$\boxed{\mathsf{viewtypes}_{\mathcal{S}} \equiv \mathrm{K}; \Gamma}$$

$$\frac{}{\begin{aligned}\mathsf{viewtypes}_{\mathcal{S}} \equiv\ & \mathrm{Zero} :: \star,\\ & \mathrm{Unit} :: \star,\\ & \mathrm{Sum} :: \star \to \star \to \star,\\ & \mathrm{Prod} :: \star \to \star \to \star;\\ & \mathit{Unit} :: \mathrm{Unit},\\ & \mathit{Inl} \ :: \forall \mathrm{a} :: \star \,.\, \forall \mathrm{b} :: \star \,.\, \mathrm{a} \to \mathrm{Sum\ a\ b},\\ & \mathit{Inr} \ :: \forall \mathrm{a} :: \star \,.\, \forall \mathrm{b} :: \star \,.\, \mathrm{b} \to \mathrm{Sum\ a\ b},\\ & (\times) \ :: \forall \mathrm{a} :: \star \,.\, \forall \mathrm{b} :: \star \,.\, \mathrm{a} \to \mathrm{b} \to \mathrm{Prod\ a\ b}\end{aligned}} \quad \text{(vt-std)}$$

Figure 6.1: View types for the standard view

discussed in terms of the core language of Chapter 2 and the formal definition of a generic view, that appears in Chapter 5.

View Types

The view types of the standard view $\mathcal{S}$ (see Figure 6.1) are given by the following declarations:

$$\begin{aligned}&\textbf{data}\ \mathrm{Zero} =\\ &\textbf{data}\ \mathrm{Unit} = \mathit{Unit}\\ &\textbf{data}\ \mathrm{Sum} = \Lambda \mathrm{a} :: \star \,.\, \Lambda \mathrm{b} :: \star \,.\, \mathit{Inl}\ \mathrm{a} \mid \mathit{Inr}\ \mathrm{b}\\ &\textbf{data}\ \mathrm{Prod} = \Lambda \mathrm{a} :: \star \,.\, \Lambda \mathrm{b} :: \star \,.\, \mathrm{a} \times \mathrm{b}.\end{aligned}$$

These types, respectively, represent nullary sums, nullary products, binary sums, and binary products.

Nullary sums are used to represent constructorless types (cf. Section 2.4). (In fact, the view type Zero is itself a constructorless type.) Although they are not valid in Haskell 98, we assume here that it is indeed allowed to declare such types. Nullary sums play a more prominent rôle in the discussion of the list-like view in Section 6.3.

Generating Structure Types

The algorithm that generates structural representations for data types is expressed by judgements of the forms

$$\begin{aligned}&\mathcal{S}[\![D_0]\!]^{\mathrm{str}} && \equiv u; \{D_i\}_{,}^{i \in 1..n}\\ &\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} && \equiv t.\end{aligned}$$

The former express how type declarations are mapped to parameterized types and lists of supporting declarations; the latter express how a type is derived from a list of constructors. The rules are shown in Figures 6.2 and 6.3.

$$\boxed{\mathcal{S}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i\in 1..n}}$$

$$\frac{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}{\begin{array}{r}\mathcal{S}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \\ \equiv \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ t; \varepsilon\end{array}} \quad \text{(str-std-data)}$$

Figure 6.2: Structural representation of data types in the standard view

$$\boxed{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}$$

$$\frac{}{\mathcal{S}[\![\varepsilon]\!]^{\mathrm{str}} \equiv \mathrm{Zero}} \quad \text{(str-std-con-1)} \qquad \frac{}{\mathcal{S}[\![C]\!]^{\mathrm{str}} \equiv \mathrm{Unit}} \quad \text{(str-std-con-2)}$$

$$\frac{}{\mathcal{S}[\![C\ t]\!]^{\mathrm{str}} \equiv t} \quad \text{(str-std-con-3)}$$

$$\frac{n \in 2..\qquad \mathcal{S}[\![C\ \{t_k\}^{k\in 2..n}]\!]^{\mathrm{str}} \equiv t'_2}{\mathcal{S}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{str}} \equiv \mathrm{Prod}\ t_1\ t'_2} \quad \text{(str-std-con-4)}$$

$$\frac{\begin{array}{c}m \in 2..\qquad \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 2..m}]\!]^{\mathrm{str}} \equiv t_2 \\ \mathcal{S}[\![C_1\ \{t_{1,k}\}^{k\in 1..n_1}]\!]^{\mathrm{str}} \equiv t_1\end{array}}{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_1\ t_2} \quad \text{(str-std-con-5)}$$

Figure 6.3: Structural representation of constructors in the standard view

$$\boxed{\mathcal{S}[\![D]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}}}$$

$$\frac{\begin{array}{c}\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m} \\ e_{\mathrm{from}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{from},j} \to p_{\mathrm{to},j}\}_{;}^{j\in 1..m} \\ e_{\mathrm{to}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{to},j} \to p_{\mathrm{from},j}\}_{;}^{j\in 1..m}\end{array}}{\begin{array}{r}\mathcal{S}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \\ \equiv e_{\mathrm{from}}; e_{\mathrm{to}}\end{array}} \quad \text{(conv-std-data)}$$

Figure 6.4: Conversions for data types in the standard view

Type declarations are handled by the rule (str-std-data). The type parameters of a declared type constructor are directly copied to the resulting structure type. Notice that the standard view never needs any auxiliary declarations.

For constructors, we distinguish five cases. The first, applying rule (str-std-con-1), deals with empty constructor lists. In that case, the nullary sum representation Zero is emitted.

The next three cases handle singleton lists of constructors. Fieldless constructors are, by rule (str-std-con-2), represented by nullary products. Rule (str-std-con-3) applies to unary constructors; for these, the sole field types are used as the structure types. If a constructor has two or more fields, rule (str-std-con-4) prescribes the recursive generation of a product type.

Finally, lists that contain two or more constructors are handled by rule (str-std-con-5). In this case, we recursively create a sum structure.

Generating Conversions

The rules for generating conversion functions are shown in Figures 6.4 and 6.5 and are of the forms

$$\begin{array}{ll}\mathcal{S}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} & \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m},\end{array}$$

i.e., type declarations give rise to pairs of conversion functions, while lists of data constructors give rise to pairs of patterns.

The rule (conv-std-data) constructs a 'from' function that matches values of the orginal type aginst a list of patterns. If a value is successfully matched against a certain pattern, a structure value is produced by using a complementary pattern; hence, here, we make use of the fact that the pattern language is just a subset of the expression language. A 'to' function is created by simply inverting the patterns.

The pairs of pattern lists are generated using the rules for constructor lists. These rules are analoguous to the rules for generating structure types from constructor lists.

$$\boxed{\mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_{|}^{j\in 1..m}; \{p_{\text{to},j}\}_{|}^{j\in 1..m}}$$

$$\frac{}{\mathcal{S}[\![\varepsilon]\!]^{\text{conv}} \equiv \varepsilon;\varepsilon} \quad \text{(conv-std-con-1)} \qquad \frac{}{\mathcal{S}[\![C]\!]^{\text{conv}} \equiv C; \mathit{Unit}} \quad \text{(conv-std-con-2)}$$

$$\frac{}{\mathcal{S}[\![C\ t]\!]^{\text{conv}} \equiv C\ x; x} \quad \text{(conv-std-con-3)}$$

$$\frac{n \in 2..\qquad \{x_1\ \not\equiv x_k\}^{k\in 2..n} \qquad \mathcal{S}[\![C\ \{t_k\}^{k\in 2..n}]\!]^{\text{conv}} \equiv C\ \{x_k\}^{k\in 2..n}; p_{\text{to}}}{\mathcal{S}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\text{conv}} \equiv C\ \{x_k\}^{k\in 1..n}; x_1 \times p_{\text{to}}} \quad \text{(conv-std-con-4)}$$

$$\frac{\begin{array}{c} m \in 2.. \\ \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 2..m}]\!]^{\text{conv}} \equiv \{p_{\text{from},j}\}_{|}^{j\in 2..m}; \{p_{\text{to},j}\}_{|}^{j\in 2..m} \\ \mathcal{S}[\![C_1\ \{t_{1,k}\}^{k\in 1..n_1}]\!]^{\text{conv}} \equiv p_{\text{from},1}; p_{\text{to},1} \end{array}}{\begin{array}{c} \mathcal{S}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\text{conv}} \\ \equiv \{p_{\text{from},j}\}_{|}^{j\in 1..m}; \mathit{Inl}\ p_{\text{to},1}\ \{|\ \mathit{Inr}\ p_{\text{to},j}\}^{j\in 2..m} \end{array}} \quad \text{(conv-std-con-5)}$$

Figure 6.5: Conversions for constructors in the standard view

If there are no constructors, rule (conv-std-con-1) prescribes that there are no patterns either.

In rule (conv-std-con-2), a single constuctor is associated with the value *Unit*. Rule (conv-std-con-3) associates unary constructors with variables that correspond to their field values. If a constructor has two or more fields, rule (conv-std-con-4) associates the corresponding variables to product patterns.

If the list of constructors has two or more elements, rule (conv-std-con-5) applies; it prefixes the patterns with the injection constructors *Inl* and *Inr*.

6.2 Balanced Encoding

In the standard view on data types the structure of a type is viewed using nested right-deep binary sums and products. The choice for such a view is rather aribrary: a nested left-deep view or a balanced view may be just as suitable. However, for some applications, the chosen view may have some impact on the behaviour of generic programs.

Generic vs. Manual Encoding

Recall the generic data-compression function *encode* from Chapter 4,

$$
\begin{array}{ll}
\textbf{data}\ \mathrm{Bit} & = \mathtt{0} \mid \mathtt{1} \\
\textbf{type}\ \mathrm{Bin} & = [\mathrm{Bit}]. \\
\\
\mathit{encode}\ \langle \mathrm{a} :: \star \rangle & :: (\mathit{encode}\ \langle \mathrm{a} \rangle) \Rightarrow \mathrm{a} \to \mathrm{Bin} \\
\mathit{encode}\ \langle \mathrm{Unit} \rangle\ \mathit{Unit} & = [\,] \\
\mathit{encode}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inl}\ a) & = \mathtt{0} : \mathit{encode}\ \langle \alpha \rangle\ a \\
\mathit{encode}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ (\mathit{Inr}\ b) & = \mathtt{1} : \mathit{encode}\ \langle \beta \rangle\ b \\
\mathit{encode}\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (a \times b) & = \mathit{encode}\ \langle \alpha \rangle\ a +\!\!+ \mathit{encode}\ \langle \beta \rangle\ b \\
\mathit{encode}\ \langle \mathrm{Integer} \rangle\ n & = \mathit{encodeInteger}\ n \\
\mathit{encode}\ \langle \mathrm{Char} \rangle\ c & = \mathit{encodeChar}\ c \\
\mathit{encode}\ \langle \mathrm{Float} \rangle\ e & = \mathit{encodeFloat}\ e,
\end{array}
$$

and consider the type Day for representing a day of the week:

$$
\begin{array}{ll}
\textbf{data}\ \mathrm{Day} = & \mathit{Sunday} \\
\mid & \mathit{Monday} \\
\mid & \mathit{Tuesday} \\
\mid & \mathit{Wednesday} \\
\mid & \mathit{Thursday} \\
\mid & \mathit{Friday} \\
\mid & \mathit{Saturday}.
\end{array}
$$

In the standard view, the structure type for Day is given by

$$
\begin{array}{l}
\textbf{type}\ \mathrm{Day}^{\circ} = \mathrm{Sum\ Unit} \\
\qquad\qquad (\mathrm{Sum\ Unit} \\
\qquad\qquad\quad (\mathrm{Sum\ Unit} \\
\qquad\qquad\qquad (\mathrm{Sum\ Unit} \\
\qquad\qquad\qquad\quad (\mathrm{Sum\ Unit} \\
\qquad\qquad\qquad\qquad (\mathrm{Sum\ Unit}
\end{array}
$$

Sunday	[0]
Monday	[1,0]
Tuesday	[1,1,0]
Wednesday	[1,1,1,0]
Thursday	[1,1,1,1,0]
Friday	[1,1,1,1,1,0]
Saturday	[1,1,1,1,1,1]

Table 6.1: Generic encoding of days of the week

Sunday	[0,0]
Monday	[0,1,0]
Tuesday	[0,1,1]
Wednesday	[1,0,0]
Thursday	[1,0,1]
Friday	[1,1,0]
Saturday	[1,1,1]

Table 6.2: Manual encoding of days of the week

(Sum Unit)))))).

Table 6.1 shows the bit-list representations that *encode* yields for the constructors of Day. The resulting encoding takes one bit at best (*Sunday*) and six bits at worst (*Friday* and *Saturday*). Obviously, we can do much better with a manually written encoder. For example, the encoding depicted in Table 6.2 takes two bits at best (*Sunday*) and three bits at worst (*Monday*, *Tuesday*, *Wednesday*, *Thursday*, *Friday*, and *Saturday*). On average, the manual encoding is much more efficient.

Balancing Sums and Products

In the balanced view $\mathcal{B}$ data types are encoded as sums and products according to a binary subdivision scheme. For instance, the structure type that $\mathcal{B}$ yields for Day is given by

$$\textbf{type}\ \mathrm{Day}^{\circ}_{\mathcal{B}} = \mathrm{Sum}\ (\mathrm{Sum\ Unit}\ \ (\mathrm{Sum\ Unit\ Unit}))\ (\mathrm{Sum}\ (\mathrm{Sum\ Unit\ Unit})\ (\mathrm{Sum\ Unit\ Unit})).$$

Note that the nesting level of sums is at most three. In general, in the balanced view, the maximum nesting level of sums in the representation of a type with n constructors is $\lfloor n/2 \rfloor$.

The next step is to let the generic *encode* function make use of the balanced view. This is achieved by simply decorating the type index in the type signature of *encode* with a *view annotation*:

$$\mathit{encode}\ \langle \mathrm{a} :: \star \leftarrow \mathcal{B} \rangle :: (\mathit{encode}\ \langle \mathrm{a} \rangle) \Rightarrow \mathrm{a} \rightarrow \mathrm{Bin}.$$

Now, applying *encode* to Day yields the same efficient encoding as a manually written function. In general, encoding a value of a type with n data construc-

$$\boxed{\mathsf{viewtypes}_{\mathcal{B}} \equiv \mathrm{K}; \Gamma}$$

$$\frac{}{\begin{aligned}\mathsf{viewtypes}_{\mathcal{B}} \equiv\ & \mathrm{Zero} :: \star,\\ & \mathrm{Unit} :: \star,\\ & \mathrm{Sum} :: \star \to \star \to \star,\\ & \mathrm{Prod} :: \star \to \star \to \star;\\ & \mathit{Unit} :: \mathrm{Unit},\\ & \mathit{Inl} :: \forall \mathrm{a} :: \star\ .\ \forall \mathrm{b} :: \star\ .\ \mathrm{a} \to \mathrm{Sum\ a\ b},\\ & \mathit{Inr} :: \forall \mathrm{a} :: \star\ .\ \forall \mathrm{b} :: \star\ .\ \mathrm{b} \to \mathrm{Sum\ a\ b},\\ & (\times) :: \forall \mathrm{a} :: \star\ .\ \forall \mathrm{b} :: \star\ .\ \mathrm{a} \to \mathrm{b} \to \mathrm{Prod\ a\ b}\end{aligned}} \quad \text{(vt-bal)}$$

Figure 6.6: View types for the balanced view

$$\boxed{\mathcal{B}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n}}$$

$$\frac{\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} \equiv t}{\begin{aligned}\mathcal{B}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}}\\ \equiv \{\Lambda a_i :: \kappa_i\ .\ \}^{i \in 1..\ell}\ t; \varepsilon\end{aligned}} \quad \text{(str-bal-data)}$$

Figure 6.7: Structural representation of data types in the balanced view

tors now requires $O(\log n)$ bits, as opposed to the $O(n)$ bits required by the standard-view version of *encode*.

If no explicit view annotation is given for a generic function, we assume it makes use of the standard view $\mathcal{S}$.

It remains to formally define the structure-type and conversion generating algorithms for the balanced view.

View Types

Figure 6.6 shows the view types of the balanced view; typesetting it did not cost the author too much effort, since it only involved copying Figure 6.1, replacing the two occurrences of $\mathcal{S}$ with $\mathcal{B}$, and modifying the rule name. Indeed, the view types of the standard view and the balanced view coincide.

Generating Structure Types

Structure types for the balanced view are generated according to rules of the forms

$$\boxed{\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}$$

$$\frac{}{\mathcal{B}[\![\varepsilon]\!]^{\mathrm{str}} \equiv \mathrm{Zero}}\ \text{(str-bal-con-1)} \qquad \frac{}{\mathcal{B}[\![C]\!]^{\mathrm{str}} \equiv \mathrm{Unit}}\ \text{(str-bal-con-2)}$$

$$\frac{}{\mathcal{B}[\![C\ t]\!]^{\mathrm{str}} \equiv t}\ \text{(str-bal-con-3)}$$

$$\frac{\begin{array}{c} n \in 2.. \qquad \mathcal{B}[\![C\ \{t_k\}^{k\in 1..\lfloor n/2\rfloor}]\!]^{\mathrm{str}} \equiv t'_1 \\ \mathcal{B}[\![C\ \{t_k\}^{k\in \lfloor n/2\rfloor+1..n}]\!]^{\mathrm{str}} \equiv t'_2 \end{array}}{\mathcal{B}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{str}} \equiv \mathrm{Prod}\ t'_1\ t'_2}\ \text{(str-bal-con-4)}$$

$$\frac{\begin{array}{c} m \in 2.. \qquad \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..\lfloor m/2\rfloor}]\!]^{\mathrm{str}} \equiv t_1 \\ \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in \lfloor m/2\rfloor+1..n}]\!]^{\mathrm{str}} \equiv t_2 \end{array}}{\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_1\ t_2}\ \text{(str-bal-con-5)}$$

Figure 6.8: Structural representation of constructors in the balanced view

$$\begin{array}{ll} \mathcal{B}[\![D_0]\!]^{\mathrm{str}} & \equiv u; \{D_i\}_{,}^{i\in 1..n} \\ \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} & \equiv t. \end{array}$$

shown in Figures 6.7 and 6.8.

The rules (str-bal-data), (str-bal-con-1), (str-bal-con-2), and (str-bal-con-3) do not differ from the corresponding rules for the standard view. The interesting bits of the structure-type generation for the balanced view are contained in the rules (str-bal-con-4) and (str-bal-con-5). These rules deal with the cases in which there are more than two fields or more than two constructors. In these cases, lists of fields or constructors are broken into pairs of lists of almost equal length, after which products or sums are generated recursively.

Generating Conversions

The rules for generating conversion functions for the balanced view are presented in Figures 6.9 and 6.10. The judgements are of the forms

$$\begin{array}{ll} \mathcal{B}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \\ \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} & \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m}. \end{array}$$

Just as it is the case for the structure-type generating algorithm, most rules closely mimic the rules for the standard view: (conv-bal-data), (conv-bal-con-1), (conv-bal-con-2), and (conv-bal-con-3). The divide-and-conquer nature of the balanced view is expressed by the rules (conv-bal-con-4) and (conv-bal-con-5).

$$\boxed{\mathcal{B}[\![D]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}}}$$

$$\frac{\begin{array}{c}\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m} \\ e_{\mathrm{from}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{from},j} \to p_{\mathrm{to},j}\}_{;}^{j\in 1..m} \\ e_{\mathrm{to}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{to},j} \to p_{\mathrm{from},j}\}_{;}^{j\in 1..m}\end{array}}{\begin{array}{r}\mathcal{B}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \\ \equiv e_{\mathrm{from}}; e_{\mathrm{to}}\end{array}} \quad \text{(conv-bal-data)}$$

Figure 6.9: Conversions for data types in the balanced view

$$\boxed{\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m}}$$

$$\frac{}{\mathcal{B}[\![\varepsilon]\!]^{\mathrm{conv}} \equiv \varepsilon; \varepsilon} \quad \text{(conv-bal-con-1)} \qquad \frac{}{\mathcal{B}[\![C]\!]^{\mathrm{conv}} \equiv C; \mathit{Unit}} \quad \text{(conv-bal-con-2)}$$

$$\frac{}{\mathcal{B}[\![C\ t]\!]^{\mathrm{conv}} \equiv C\ x; x} \quad \text{(conv-bal-con-3)}$$

$$\frac{\begin{array}{c}n \in 2..\qquad \{\{x_k \not\equiv x_{k'}\}^{k'\in\lfloor n/2\rfloor+1..n}\}^{k\in 1..\lfloor n/2\rfloor} \\ \mathcal{B}[\![C\ \{t_k\}^{k\in 1..\lfloor n/2\rfloor}]\!]^{\mathrm{conv}} \equiv C\ \{x_k\}^{k\in 1..\lfloor n/2\rfloor}; p_{\mathrm{to},1} \\ \mathcal{B}[\![C\ \{t_k\}^{k\in\lfloor n/2\rfloor+1..n}]\!]^{\mathrm{conv}} \equiv C\ \{x_k\}^{k\in\lfloor n/2\rfloor+1..n}; p_{\mathrm{to},2}\end{array}}{\mathcal{B}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}} \equiv C\ \{x_k\}^{k\in 1..n}; p_{\mathrm{to},1} \times p_{\mathrm{to},2}} \quad \text{(conv-bal-con-4)}$$

$$\frac{\begin{array}{c}m \in 2.. \\ \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..\lfloor m/2\rfloor}]\!]^{\mathrm{conv}} \\ \quad \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..\lfloor m/2\rfloor}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..\lfloor m/2\rfloor} \\ \mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in\lfloor m/2\rfloor+1..m}]\!]^{\mathrm{conv}} \\ \quad \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in\lfloor m/2\rfloor+1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in\lfloor m/2\rfloor+1..m}\end{array}}{\begin{array}{l}\mathcal{B}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \\ \quad \equiv\ \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \\ \qquad \{\mathit{Inl}\ p_{\mathrm{to},j}\}_{|}^{j\in 1..\lfloor m/2\rfloor}\ \{|\ \mathit{Inr}\ p_{\mathrm{to},j}\}^{j\in\lfloor m/2\rfloor+1..m}\end{array}} \quad \text{(conv-bal-con-5)}$$

Figure 6.10: Conversions for constructors in the balanced view

6.3 List-like Encoding

The balanced view on data types is an excellent example of how the chosen view for a generic function can determine its behaviour. In this section, we examine a view that allows functions to be expressed that are difficult to write in, for example, the standard view.

Partial Pretty Printing

Suppose we have a generic function *show*,

$$show\ \langle \mathrm{a} :: \star \rangle :: (show\ \langle \mathrm{a} \rangle) \Rightarrow \mathrm{a} \to \mathrm{String},$$

that pretty prints values of any data type. We want to use this function to define a generic function *showPart* that prints only a part of a given value. Which part is printed is determined by an argument of the type Path of paths,

$$\textbf{type}\ \mathrm{Path} = [\mathrm{Integer}].$$

A non-empty value of type Path selects a certain node within a data structure. For instance, the value [1] selects the second field of the structure's top-level data constructor; the value [1, 0] selects the first field of the top-level constructor thereof. An empty path selects the complete data structure.

The following is a first attempt to define the function *showPart* using the standard view on types:

$$\begin{array}{ll}
showPart\ \langle \mathrm{a} :: \star \rangle & :: \\
\quad (showPart\ \langle \mathrm{a} \rangle, show\ \langle \mathrm{a} \rangle) \Rightarrow \mathrm{Path} \to \mathrm{a} \to \mathrm{String} & \\
showPart\ \langle \mathrm{Unit} \rangle\ p\ Unit & = error\ \texttt{"showPart"} \\
showPart\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ [\,]\ x & = show\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ x \\
showPart\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ p\ (Inl\ a) & = showPart\ \langle \alpha \rangle\ p\ a \\
showPart\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle\ p\ (Inr\ b) & = showPart\ \langle \beta \rangle\ p\ b \\
showPart\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ [\,]\ (a \times b) & = error\ \texttt{"showPart"} \\
showPart\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (0 : [\,])\ (a \times b) & = show\ \langle \alpha \rangle\ a \\
showPart\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (0 : p)\ (a \times b) & = showPart\ \langle \alpha \rangle\ p\ a \\
showPart\ \langle \mathrm{Prod}\ \alpha\ \beta \rangle\ (n : p)\ (a \times b) & = showPart\ \langle \beta \rangle\ (n-1 : p)\ b \\
showPart\ \langle \mathrm{Integer} \rangle\ [\,]\ n & = show\ \langle \mathrm{Integer} \rangle\ n \\
showPart\ \langle \mathrm{Integer} \rangle\ p\ n & = error\ \texttt{"showPart"} \\
showPart\ \langle \mathrm{Char} \rangle\ [\,]\ c & = show\ \langle \mathrm{Char} \rangle\ c \\
showPart\ \langle \mathrm{Char} \rangle\ p\ c & = error\ \texttt{"showPart"} \\
showPart\ \langle \mathrm{Float} \rangle\ [\,]\ e & = show\ \langle \mathrm{Float} \rangle\ e \\
showPart\ \langle \mathrm{Float} \rangle\ p\ e & = error\ \texttt{"showPart"}.
\end{array}$$

In the case for Unit there is not much we can do. If the path is non-empty, we are stuck, since there is no way to further navigate into a nullary product. Furthermore, we assume that the situation in which the path in empty is taken care of on the level of sums; that way, we do not have to reproduce the code that handles empty paths for both nullary and binary products. Thus, if *showPart* is called for a sum type and an empty path, printing of the value is delegated to the generic *show* function. If *showPart* is called for a sum type and a non-empty path, we recursively call the *showPart* function on the appropriate type and

value. For products we distinguish four cases. First, if the path is empty, we have to raise an error, because we assume this case is handled by the sum case. Second, if the part is non-empty and the only path element is 0, the leftmost component of the product is printed. Third, if the path contains two or more elements and the first element is 0, we recurse into the first component of the product. And, fourth, if the first element of the path is not 0, we decrease it and recurse into the second component of the product. Finally, calls to *showPart* for abstract types can only be handled if the paths are empty.

Although this definition may seem fine at first sight, we still have to consider a few problematic degenerate cases.

Degenerate Cases

Consider the call

$$\mathit{showPart}\ \langle \text{Fork Integer} \rangle\ [\,]\ (\mathit{Fork}\ 2\ 3),$$

which is supposed to pretty print the complete value *Fork* 2 3. Since the standard view represents the type Fork Integer by Prod Integer Integer, this call translates to

$$\mathit{showPart}\ \langle \text{Prod Integer Integer} \rangle\ [\,]\ (2 \times 3).$$

However, a quick look the definition of *showPart* reveals that this yields an error. Indeed, we assumed that the case for empty paths was taken care of at the level of sums. Yet the structure type for Fork does not contain a sum! To circumvent this problem we could hide the call to *showPart* behind a generic abstraction *showPart′* (cf. Section 4.3),

$$\begin{aligned}
&\mathit{showPart'}\ \langle \text{a} :: \star \rangle \qquad :: (\mathit{showPart}\ \langle \text{a} \rangle, \mathit{show}\ \langle \text{a} \rangle) \Rightarrow \text{Path} \to \text{a} \to \text{String} \\
&\mathit{showPart'}\ \langle \alpha :: \star \rangle\ [\,]\ a = \mathit{show}\ \langle \alpha \rangle\ a \\
&\mathit{showPart'}\ \langle \alpha :: \star \rangle\ p\ a\ \ = \mathit{showPart}\ \langle \alpha \rangle\ p\ a,
\end{aligned}$$

and replace the recursive calls in *showPart* by calls to *showPart′*. Alternatively, we could of course act on empty paths in the cases for nullary and binary products—but then we would distribute a single implementation aspect over multiple sites, which is, in general, not desirable.

Another problem arises when a structure does not contain a product—i.e., for unary constructors like *Tip* of Tree. For example, the call

$$\mathit{showPart}\ \langle \text{Tree Integer Bool} \rangle\ [0]\ (\mathit{Tip}\ 5)$$

should pretty print the value 5, but yields an error for it translates to

$$\mathit{showPart}\ \langle \text{Integer} \rangle\ [0]\ 5.$$

A solution would be to add a dependency to another generic function—one that is capable of detecting the absence of a product structure—and call this function from the cases for sums. But then, of course, we should also modify the implementation of the generic abstraction *showPart′*—to handle cases in which there is no sum structure.

In the end, we obtain a rather complicated implementation of *showPart*.

Lists of Terms, Lists of Factors

The problems discussed in the previous subsection are all caused by the irregular nature of the structural representations generated by the standard view: one cannot rely on the presence of sum and product structures. Therefore, we introduce a more regular encoding in sums and products.

In the list-like view $\mathcal{L}$, the left component of a sum always encodes a single constructor, while the right component is always another—nullary or binary—sum. Likewise, the left component of a product always encodes a single field, while the right component is always another product. Moreover, each structural representation contains a sum and a product structure. Sums are presented as *Zero*-terminated lists of terms, products as Unit-terminated lists of factors.

For instance, the types Fork and Tree,

```
data Fork (a :: ⋆)         = Fork a a
data Tree (a :: ⋆) (b :: ⋆) = Tip a | Node (Tree a b) b (Tree a b),
```

are encoded as

```
type Fork°_L (a :: ⋆)         = Sum (Prod a (Prod a Unit)) Zero
type Tree°_L (a :: ⋆) (b :: ⋆) =
  Sum (Prod a Unit)
    (Sum (Prod (Tree a b) (Prod b (Prod (Tree a b) Unit)))
      Zero).
```

Compared to the representations in the standard view, these types are rather verbose, but their structure is much more regular.

Using the list-like view $\mathcal{L}$, the function *showPart* can be defined as follows:

```
showPart ⟨a :: ⋆ ← L⟩                    ::
  (showPart ⟨a⟩, show ⟨a⟩) ⇒ Path → a → String
showPart ⟨Unit⟩ p Unit                   = error "showPart"
showPart ⟨Sum α β⟩ [] x                  = show ⟨Sum α β⟩ x
showPart ⟨Sum α β⟩ p (Inl a)             = showPart ⟨α⟩ p a
showPart ⟨Sum α β⟩ p (Inr b)             = showPart ⟨β⟩ p b
showPart ⟨Prod α β⟩ (0 : p) (a × b)      = showPart ⟨α⟩ p a
showPart ⟨Prod α β⟩ (n : p) (a × b)      = showPart ⟨β⟩ (n − 1 : p) b
showPart ⟨Integer⟩ [] n                  = show ⟨Integer⟩ n
showPart ⟨Integer⟩ p n                   = error "showPart"
showPart ⟨Char⟩ [] c                     = show ⟨Char⟩ c
showPart ⟨Char⟩ p c                      = error "showPart"
showPart ⟨Float⟩ [] e                    = show ⟨Float⟩ e
showPart ⟨Float⟩ p e                     = error "showPart".
```

If we ignore the cases for abstract types, the only check for the empty path occurs in the sum case. This can be done, because the list-like view guarantees that every type has a sum structure. The error that is raised in the case for nullary products can only occur if an invalid path is passed to the function.

As an example, consider the problematic instances from the previous subsection:

$$\boxed{\mathsf{viewtypes}_{\mathcal{L}} \equiv \mathrm{K}; \Gamma}$$

$$\frac{}{\begin{aligned}\mathsf{viewtypes}_{\mathcal{L}} \equiv\ & \mathrm{Zero} :: \star,\\ & \mathrm{Unit} :: \star,\\ & \mathrm{Sum} :: \star \to \star \to \star,\\ & \mathrm{Prod} :: \star \to \star \to \star;\\ & \mathit{Unit} :: \mathrm{Unit},\\ & \mathit{Inl} :: \forall \mathrm{a} :: \star\,.\, \forall \mathrm{b} :: \star\,.\, \mathrm{a} \to \mathrm{Sum\ a\ b},\\ & \mathit{Inr} :: \forall \mathrm{a} :: \star\,.\, \forall \mathrm{b} :: \star\,.\, \mathrm{b} \to \mathrm{Sum\ a\ b},\\ & (\times) :: \forall \mathrm{a} :: \star\,.\, \forall \mathrm{b} :: \star\,.\, \mathrm{a} \to \mathrm{b} \to \mathrm{Prod\ a\ b}\end{aligned}} \quad \text{(vt-list)}$$

Figure 6.11: View types for the list-like view

$$\begin{aligned}&\mathit{showPart}\ \langle \mathrm{Fork\ Integer} \rangle\ [\,]\ (\mathit{Fork}\ 2\ 3)\\ &\mathit{showPart}\ \langle \mathrm{Tree\ Integer\ Bool} \rangle\ [0]\ (\mathit{Tip}\ 5).\end{aligned}$$

These calls now translate to

$$\begin{aligned}&\mathit{show}\ \langle \mathrm{Sum\ (Prod\ Integer\ (Prod\ Integer\ Unit))}\ \mathit{Zero} \rangle\\ &\quad (\mathit{Inl}\ (2 \times (3 \times \mathit{Unit})))\\ &\mathit{show}\ \langle \mathrm{Integer} \rangle\ 5\end{aligned}$$

and produce the expected output[1].

View Types

The view types for the list-like view $\mathcal{L}$ are shown in Figure 6.11. Again, these types are the same as the ones for the standard view.

Generating Structure Types

The rules for generating structure types (Figures 6.12 and 6.13) take the forms

$$\begin{aligned}&\mathcal{L}[\![D_0]\!]^{\mathrm{str}} && \equiv u; \{D_i\}_{,}^{i \in 1..n}\\ &\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}]\!]^{\mathrm{str}} && \equiv t.\end{aligned}$$

As usual, the rule for type declarations, (str-list-data), delegates most of the work to the rules for constructors.

Rule (str-list-con-1) simply emits the empty sum list *Zero* for empty constructor lists. The rules that deal with singleton constructor lists—(str-list-con-2), (str-list-con-3), and (str-list-con-4)—produce singleton sum lists, i.e., types of the form Sum *t Zero*. For fieldless constructors, the rule (str-list-con-2) emits

[1] We assume that the function *show* is equipped to handle the more regular representations of the list-like view.

$$\boxed{\mathcal{L}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i\in 1..n}}$$

$$\frac{\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}{\begin{array}{r}\mathcal{L}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \\ \equiv \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ t; \varepsilon\end{array}} \quad \text{(str-list-data)}$$

Figure 6.12: Structural representation of data types in the list-like view

$$\boxed{\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv t}$$

$$\frac{}{\mathcal{L}[\![\varepsilon]\!]^{\mathrm{str}} \equiv \mathrm{Zero}} \quad \text{(str-list-con-1)}$$

$$\frac{}{\mathcal{L}[\![C]\!]^{\mathrm{str}} \equiv \mathrm{Sum\ Unit\ Zero}} \quad \text{(str-list-con-2)}$$

$$\frac{}{\mathcal{L}[\![C\ t]\!]^{\mathrm{str}} \equiv \mathrm{Sum\ (Prod}\ t\ \mathrm{Unit)\ Zero}} \quad \text{(str-list-con-3)}$$

$$\frac{n \in 2..\qquad \mathcal{L}[\![C\ \{t_k\}^{k\in 2..n}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_2'\ \mathrm{Zero}}{\mathcal{L}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{str}} \equiv \mathrm{Sum\ (Prod}\ t_1\ t_2')\ \mathrm{Zero}} \quad \text{(str-list-con-4)}$$

$$\frac{\begin{array}{l}m \in 2..\qquad \mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 2..m}]\!]^{\mathrm{str}} \equiv t_2 \\ \mathcal{L}[\![C_1\ \{t_{1,k}\}^{k\in 1..n_1}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_1\ \mathrm{Zero}\end{array}}{\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{str}} \equiv \mathrm{Sum}\ t_1\ t_2} \quad \text{(str-list-con-5)}$$

Figure 6.13: Structural representation of constructors in the list-like view

$$\boxed{\mathcal{L}[\![D]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}}}$$

$$\frac{\begin{array}{c}\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m} \\ e_{\mathrm{from}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{from},j} \to p_{\mathrm{to},j}\}_{;}^{j\in 1..m} \\ e_{\mathrm{to}} \equiv \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of}\ \{p_{\mathrm{to},j} \to p_{\mathrm{from},j}\}_{;}^{j\in 1..m}\end{array}}{\begin{array}{r}\mathcal{L}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \\ \equiv e_{\mathrm{from}}; e_{\mathrm{to}}\end{array}} \quad \text{(conv-list-data)}$$

Figure 6.14: Conversions for data types in the list-like view

the empty product list Unit. Unary constructors are handled by (str-list-con-3) and are represented by singleton product lists: types that take the form Prod t Unit. The rules (str-list-con-4) and (str-list-con-5) essentially perform list concatenation on, respectively, product and sum lists.

Generating Conversions

Figures 6.14 and 6.15 depict the rules for generating conversion functions; these have the forms

$$\begin{array}{ll}\mathcal{L}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \\ \mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} & \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m}.\end{array}$$

The rule (conv-list-data) simply assembles the functions from the patterns that are generated by the rules for constructor lists.

Rule (conv-list-con-1) maps an empty list of constructors to two empty lists of patterns. The rules (conv-list-con-2), (conv-list-con-2), and (conv-list-con-3) for singleton lists ensure that the relevant patterns are injected in the left component of a sum structure and that each product list is terminated by the value *Unit*. Finally, (conv-list-con-5) puts patterns in the appropriate positions of the sum lists.

6.4 Notes

In his proposal to extend Haskell with generic type-indexed functions [32], Hinze already favoured a balanced view on data types over a nested right-deep encoding, for the latter aggrevates the compression rate yielded by the generic data-compression function.

The regular behaviour of the list-like view on data types, as opposed to the standard view and the balanced view, make it an excellent choice for generic programs that directly operate on the structure of data types. One possible application area may be the generic editing of structured documents [83].

$$\boxed{\mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 1..m}}$$

$$\frac{}{\mathcal{L}[\![\varepsilon]\!]^{\mathrm{conv}} \equiv \varepsilon;\varepsilon} \quad \text{(conv-list-con-1)}$$

$$\frac{}{\mathcal{L}[\![C]\!]^{\mathrm{conv}} \equiv C; \mathit{Inl}\ \mathit{Unit}} \quad \text{(conv-list-con-2)}$$

$$\frac{}{\mathcal{L}[\![C\ t]\!]^{\mathrm{conv}} \equiv C\ x; \mathit{Inl}\ (x \times \mathit{Unit})} \quad \text{(conv-list-con-3)}$$

$$\frac{n \in 2..\qquad \{x_1\ \not\equiv x_k\}^{k\in 2..n} \qquad \mathcal{L}[\![C\ \{t_k\}^{k\in 2..n}]\!]^{\mathrm{conv}} \equiv C\ \{x_k\}^{k\in 2..n}; \mathit{Inl}\ p_{\mathrm{to}}}{\mathcal{L}[\![C\ \{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}} \equiv C\ \{x_k\}^{k\in 1..n}; \mathit{Inl}\ (x_1 \times p_{\mathrm{to}})} \quad \text{(conv-list-con-4)}$$

$$\frac{\begin{array}{c} m \in 2.. \\ \mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 2..m}]\!]^{\mathrm{conv}} \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 2..m}; \{p_{\mathrm{to},j}\}_{|}^{j\in 2..m} \\ \mathcal{L}[\![C_1\ \{t_{1,k}\}^{k\in 1..n_1}]\!]^{\mathrm{conv}} \equiv p_{\mathrm{from},1}; p_{\mathrm{to},1} \end{array}}{\begin{array}{c} \mathcal{L}[\![\{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathrm{conv}} \\ \equiv \{p_{\mathrm{from},j}\}_{|}^{j\in 1..m}; p_{\mathrm{to},1}\ \{|\ \mathit{Inr}\ p_{\mathrm{to},j}\}^{j\in 2..m} \end{array}} \quad \text{(conv-list-con-5)}$$

Figure 6.15: Conversions for constructors in the list-like view

Chapter 7

Recursive Calls in Type Declarations

In the previous chapter, three generic views on data types were discussed: the standard view, a balanced view, and a list-like view. Varying on the theme of sums and products, these views all share the same set of view types.

This chapter introduces two views that make use of other view types. These views, called the fixed-point view and the recursive view, provide access to recursive calls in type declarations and consequently allow the definitions of functions that cannot be expressed using the sums-and-products views from Chapter 6. It follows that the use of these views improve the expressiveness of our generic-programming system considerably.

7.1 Fixed-point View

Consider the function *subterms* that, given a value of the type Term of lambda terms,

$$
\begin{array}{l}
\textbf{data}\ \text{Variable} = \mathit{Variable}\ \text{String} \\
\textbf{data}\ \text{Term} \quad = \mathit{Var}\ \text{Variable} \\
\qquad\qquad\quad \mid \mathit{Abs}\ \text{Variable Term} \\
\qquad\qquad\quad \mid \mathit{App}\ \text{Term Term},
\end{array}
$$

produces the immediate subterms:

$$
\begin{array}{lcl}
\mathit{subterms} & :: & \text{Term} \to [\text{Term}] \\
\mathit{subterms}\ (\mathit{Var}\ x) & = & [\,] \\
\mathit{subterms}\ (\mathit{Abs}\ x\ t) & = & [t] \\
\mathit{subterms}\ (\mathit{App}\ t_1\ t_2) & = & [t_1, t_2].
\end{array}
$$

This function is an instance of a more general pattern. The function *subtrees*, for example, produces the immediate subtrees of an external binary search tree:

$$
\begin{array}{l}
\textbf{data}\ \text{Tree}\ (\text{a} :: \star)\ (\text{b} :: \star) = \mathit{Tip}\ \text{a} \mid \mathit{Node}\ (\text{Tree a b})\ \text{b}\ (\text{Tree a b}) \\
\mathit{subtrees} \qquad\qquad :: \forall (\text{a} :: \star)\ (\text{b} :: \star)\ .\ \text{Tree a b} \to [\text{Tree a b}]
\end{array}
$$

$$\begin{aligned} &\mathit{subtrees}\ (\mathit{Tip}\ a) &&= [\,] \\ &\mathit{subtrees}\ (\mathit{Node}\ l\ b\ r) &&= [l, r]. \end{aligned}$$

Likewise, the function *tails* retrieves, if available, the tail of a list:

$$\begin{aligned} &\mathbf{data}\ [\mathrm{a} :: \star] = [\,] \mid \mathrm{a} : [\mathrm{a}] \\ &\mathit{tails} &&:: \forall(\mathrm{a} :: \star)\ .\ [\mathrm{a}] \to [[\mathrm{a}]] \\ &\mathit{tails}\ [\,] &&= [\,] \\ &\mathit{tails}\ (a : as) &&= [as]. \end{aligned}$$

Notice that *tails* is the list-of-successes version [88] of the function *tail* from the Haskell 98 Libraries [75].

The functions *subterms*, *subtrees*, and *tails* all retrieve the immediate children of values of a recursive data type. Since the general pattern is clear, we would like to be able to express it as a generic function. However, neither the standard view nor any variation thereof allows us to define such a function directly. This is due to the fact that the structures in terms of sums and products, over which functions in the standard view are inductively defined, do not expose the recursive calls within a data type's definition.

Fixed Points

Interestingly, it is in fact possible, using the standard view, to write a generic program that produces the immediate children of a value—but it requires some extra effort from the user of the program. The idea is to express regular recursive data types in terms of a type-level fixed-point operator:

$$\mathbf{data}\ \mathrm{Fix}\ (\mathrm{f} :: \star \to \star) = \mathit{In}\ (\mathrm{f}\ (\mathrm{Fix}\ \mathrm{f})).$$

This type is, for instance, employed by Meijer and Hutton [64] to give a generic account of catamorphisms [61].

An alternative definition for the type of lambda terms can be given once the so-called *pattern functor* or *base type* is identified:

$$\begin{aligned} \mathbf{data}\ \mathrm{TermBase}\ (\mathrm{r} :: \star) = {} & \mathit{VarBase}\ \mathrm{Variable} \\ \mid {} & \mathit{AbsBase}\ \mathrm{Variable}\ \mathrm{r} \\ \mid {} & \mathit{AppBase}\ \mathrm{r}\ \mathrm{r}. \end{aligned}$$

The pattern functor TermBase differs from Term in that it takes a type parameter r; at the right-hand side of the definition, this parameter is substituted for each recursive call in Term. Applying the type-level fixed point operator Fix to the pattern functor TermBase yields an alternative type for terms:

$$\mathbf{type}\ \mathrm{Term}' = \mathrm{Fix}\ \mathrm{TermBase}.$$

For example, while the identity combinator $\lambda x\ .\ x$ is represented by the Term value

$$\mathit{Abs}\ (\mathit{Variable}\ \texttt{"x"})\ (\mathit{Var}\ (\mathit{Variable}\ (\texttt{"x"})),$$

the corresponding value of type Term′ is given by

In (*AbsBase* (*Variable* "x") (*In* (*VarBase* (*Variable* "x")))).

The types Tree and [ ] can also be redefined in terms of Fix and their pattern functors:

```
data TreeBase (a :: ⋆) (b :: ⋆) (r :: ⋆) = TipBase a | NodeBase r b r
type Tree′ (a :: ⋆) (b :: ⋆)             = Fix (TreeBase a b)

data ListBase (a :: ⋆) (r :: ⋆)          = NilBase | ConsBase a r
type List′ (a :: ⋆)                      = Fix (ListBase a).
```

In general, if a type has kind

$$\{\kappa_i \to\}^{i \in 1..\ell} \to \star,$$

its pattern functor has kind

$$\{\kappa_i \to\}^{i \in 1..\ell} \to \star \to \star.$$

If regular recursive types are defined using Fix, then, given the definition of *collect* (see Chapter 4),

```
collect ⟨a :: ⋆ | c :: ⋆⟩      :: (collect ⟨a | c⟩) ⇒ a → [c]
collect ⟨Unit⟩ Unit            = [ ]
collect ⟨Sum α β⟩ (Inl a)      = collect ⟨α⟩ a
collect ⟨Sum α β⟩ (Inr b)      = collect ⟨β⟩ b
collect ⟨Prod α β⟩ (a × b)     = collect ⟨α⟩ a ++ collect ⟨β⟩ b
collect ⟨Integer⟩ n            = [ ]
collect ⟨Char⟩ c               = [ ]
collect ⟨Float⟩ e              = [ ],
```

a type-indexed function *children* can be written with a single, special case for Fix:

```
children ⟨a :: ⋆⟩          :: (collect ⟨a | a⟩) ⇒ a → [a]
children ⟨Fix φ⟩ (In r) = let collect ⟨α⟩ a = [a]
                          in  collect ⟨φ α⟩ r.
```

The *children* function depends on *collect*. The local redefinition fixes the type of the produced list and adapts the *collect* function to construct singleton lists from the recursive components in a fixed point's value. The *collect* function concatenates these singletons to produce the resulting list of immediate children.

Towards a Fixed-point View

The described technique makes it possible to implement a *children* function using the standard view—it has an obvious downside, though: the programmer needs to redefine her recursive data types in terms of Fix. Whenever she wants to use *children* to compute the resursive components of a value if any of the original recursive types—say Term, Tree, or [ ]—a user defined bidirectional mapping from the original types to the fixed points—Term′, Tree′, and List′—has to be applied. Hence, the solution is not truly generic.

A *fixed-point view* on data types equips the generic-programming system with the capability of deriving the fixed point for any recursive data type and generating the required mappings automatically. In such a view, the structure of types is no longer perceived as a sum of products, but as a fixed point of a pattern functor.

In the remainder of this section, we discuss the fixed-point view $\mathcal{F}$. In this view, the *children* function is made generic by simply adapting its type structure:

$$\mathit{children}\ \langle \mathrm{a} :: \star \leftarrow \mathcal{F} \rangle :: (\mathit{collect}\ \langle \mathrm{a} \mid \mathrm{a} \rangle) \Rightarrow \mathrm{a} \rightarrow [\mathrm{a}].$$

Applying *children* directly to a recursive data type, for instance

$$\mathit{children}\ \langle [\mathrm{Integer}] \rangle\ [2, 3, 5],$$

then produces the desired result—$[3, 5]$.

For this to work, the view $\mathcal{F}$ automatically derives the pattern functor of a given algebraic data type; for example:

$$\begin{array}{ll}
\textbf{data}\ \mathrm{Term}^{\bullet}\ (\mathrm{r} :: \star) & = \mathit{Var}^{\bullet}\ \mathrm{Variable} \\
 & \mid \mathit{Abs}^{\bullet}\ \mathrm{Variable\ r} \\
 & \mid \mathit{App}^{\bullet}\ \mathrm{r\ r} \\
\textbf{data}\ \mathrm{Tree}^{\bullet}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star)\ (\mathrm{r} :: \star) & = \mathit{Tip}^{\bullet}\ \mathrm{a} \\
 & \mid \mathit{Node}^{\bullet}\ \mathrm{r\ b\ r} \\
\textbf{data}\ [\,]^{\bullet}\ (\mathrm{a} :: \star)\ (\mathrm{r} :: \star) & = [\,]^{\bullet} \\
 & \mid \mathrm{a} :^{\bullet} \mathrm{r}.
\end{array}$$

Then, structure types arise by applying Fix to these pattern functors:

$$\begin{array}{ll}
\textbf{type}\ \mathrm{Term}^{\circ}_{\mathcal{F}} & = \mathrm{Fix\ Term}^{\bullet} \\
\textbf{type}\ \mathrm{Tree}^{\circ}_{\mathcal{F}}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star) & = \mathrm{Fix\ (Tree}^{\bullet}\ \mathrm{a\ b)} \\
\textbf{type}\ [\mathrm{a} :: \star]^{\circ}_{\mathcal{F}} & = \mathrm{Fix\ ([\,]}^{\bullet}\ \mathrm{a}).
\end{array}$$

The fixed-point view might not be very useful to nested types, because Fix does not give access to all recursive calls in those types. However, one might choose to still include those types in the view domain, since it is indeed possible to derive pattern functors for them; for instance:

$$\begin{array}{ll}
\textbf{data}\ \mathrm{Perfect}\ (\mathrm{a} :: \star) & = \mathit{ZeroP}\ \mathrm{a} \\
 & \mid \mathit{SuccP}\ (\mathrm{Perfect\ (Fork\ a)}) \\
\textbf{data}\ \mathrm{Perfect}^{\bullet}\ (\mathrm{a} :: \star)\ (\mathrm{r} :: \star) & = \mathit{ZeroP}^{\bullet}\ \mathrm{a} \\
 & \mid \mathit{SuccP}^{\bullet}\ (\mathrm{Perfect\ (Fork}\ a)) \\
\textbf{type}\ \mathrm{Perfect}^{\circ}_{\mathcal{F}}\ (\mathrm{a} :: \star) & = \mathrm{Fix\ (Perfect}^{\bullet}\ \mathrm{a}).
\end{array}$$

So, the view $\mathcal{F}$ just ignores irregular recursive calls. For example, the call

$$\mathit{children}\ \langle \mathrm{Perfect\ Integer} \rangle\ (\mathit{SuccP}\ (\mathit{ZeroP}\ (\mathit{Fork}\ 2\ 3)))$$

yields $[\,]$.

Bidirectional Mappings

Unlike the view types for the sum-of-products views, the type Fix is recursive. As it turns out, this has major implications for the implementation of the view $\mathcal{F}$.

Let us look at the generated conversion functions for $\mathcal{F}$. For the type Term, for instance, we have

$$
\begin{array}{ll}
\mathit{fromTerm}_{\mathcal{F}} & :: \mathrm{Term} \to \mathrm{Term}^{\circ}_{\mathcal{F}} \\
\mathit{fromTerm}_{\mathcal{F}}\ (\mathit{Var}\ x) & = \mathit{In}\ (\mathit{Var}^{\bullet}\ x) \\
\mathit{fromTerm}_{\mathcal{F}}\ (\mathit{Abs}\ x\ t) & = \mathit{In}\ (\mathit{Abs}^{\bullet}\ x\ (\mathit{fromTerm}_{\mathcal{F}}\ t)) \\
\mathit{fromTerm}_{\mathcal{F}}\ (\mathit{App}\ t_1\ t_2) & = \mathit{In}\ (\mathit{App}^{\bullet}\ (\mathit{fromTerm}_{\mathcal{F}}\ t_1)\ (\mathit{fromTerm}_{\mathcal{F}}\ t_2)) \\
\\
\mathit{toTerm}_{\mathcal{F}} & :: \mathrm{Term}^{\circ}_{\mathcal{F}} \to \mathrm{Term} \\
\mathit{toTerm}_{\mathcal{F}}\ (\mathit{In}\ (\mathit{Var}^{\bullet}\ x)) & = \mathit{Var}\ x \\
\mathit{toTerm}_{\mathcal{F}}\ (\mathit{In}\ (\mathit{Abs}^{\bullet}\ x\ t)) & = \mathit{Abs}\ x\ (\mathit{toTerm}_{\mathcal{F}}\ t) \\
\mathit{toTerm}_{\mathcal{F}}\ (\mathit{In}\ (\mathit{App}^{\bullet}\ t_1\ t_2)) & = \mathit{App}\ (\mathit{toTerm}_{\mathcal{F}}\ t_1)\ (\mathit{toTerm}_{\mathcal{F}}\ t_2) \\
\\
\mathit{convTerm}_{\mathcal{F}} & :: \mathit{Conv}\ \mathrm{Term}\ \mathrm{Term}^{\circ}_{\mathcal{F}} \\
\mathit{convTerm}_{\mathcal{F}} & = \mathit{Conv}\{\mathit{from} = \mathit{fromTerm}_{\mathcal{F}}, \mathit{to} = \mathit{toTerm}_{\mathcal{F}}\}.
\end{array}
$$

Note that, as a consequence of the recursion in Fix, the functions $\mathit{fromTerm}_{\mathcal{F}}$ and $\mathit{toTerm}_{\mathcal{F}}$ are recursive.

Now consider the type Rose,

$$
\begin{array}{ll}
\mathbf{data}\ \mathrm{Rose}\ (\mathrm{a} :: \star) & = \mathit{Branch}\ \mathrm{a}\ [\mathrm{Rose\ a}] \\
\mathbf{data}\ \mathrm{Rose}^{\bullet}\ (\mathrm{a} :: \star)\ (\mathrm{r} :: \star) & = \mathit{Branch}^{\bullet}\ \mathrm{a}\ [\mathrm{r}] \\
\mathbf{type}\ \mathrm{Rose}^{\circ}_{\mathcal{F}}\ (\mathrm{a} :: \star) & = \mathrm{Fix}\ (\mathrm{Rose}^{\bullet}\ \mathrm{a}).
\end{array}
$$

Observe that the recursive call in the definition of Rose occurs in the argument position of an application of the type constructor $[\,]$. As a consequence, the conversion functions for Rose need to be mapped over list structures:

$$
\begin{array}{ll}
\mathit{fromRose}_{\mathcal{F}} & :: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Rose\ a} \to \mathrm{Rose}^{\circ}_{\mathcal{F}}\ \mathrm{a} \\
\mathit{fromRose}_{\mathcal{F}}\ (\mathit{Branch}\ a\ as) & = \mathit{In}\ (\mathit{Branch}^{\bullet}\ a\ (\mathit{mapList}\ \mathit{fromRose}_{\mathcal{F}}\ as)) \\
\\
\mathit{toRose}_{\mathcal{F}} & :: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Rose}^{\circ}_{\mathcal{F}}\ \mathrm{a} \to \mathrm{Rose\ a} \\
\mathit{toRose}_{\mathcal{F}}\ (\mathit{In}\ (\mathit{Branch}^{\bullet}\ a\ as)) & = \mathit{Branch}\ a\ (\mathit{mapList}\ \mathit{toRose}_{\mathcal{F}}\ as) \\
\\
\mathit{convRose}_{\mathcal{F}} & :: \forall(\mathrm{a} :: \star)\ .\ \mathit{Conv}\ (\mathrm{Rose\ a})\ (\mathrm{Rose}^{\circ}_{\mathcal{F}}\ \mathrm{a}) \\
\mathit{convRose}_{\mathcal{F}} & = \mathit{Conv}\{\mathit{from} = \mathit{fromRose}_{\mathcal{F}}, \mathit{to} = \mathit{toRose}_{\mathcal{F}}\}.
\end{array}
$$

In general, to be able to generate appropriate conversions for the fixed-point view, we need to have mapping functions available for any possible type. The generic mapping function *map*, discussed in Chapter 4, is not powerful enough: it cannot be defined for function types, since the type constructor $(\to)$ is contravariant in its first argument. However, this problem can be solved by using a generic bidirectional mapping function:

$$
\begin{array}{ll}
\mathit{bimap}\ \langle \mathrm{a} :: \star, \mathrm{b} :: \star \rangle & :: (\mathit{bimap}\ \langle \mathrm{a}, \mathrm{b} \rangle) \Rightarrow \mathrm{Conv\ a\ b} \\
\mathit{bimap}\ \langle \mathrm{Unit} \rangle & = \mathit{Conv}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} \\
\mathit{bimap}\ \langle \mathrm{Sum}\ \alpha\ \beta \rangle & = \\
\quad \mathbf{let}\ \mathit{fromSum}\ x = \mathbf{case}\ x\ \mathbf{of} & \\
\qquad\qquad \mathit{Inl}\ a & \to \mathit{Inl}\ (\mathit{from}\ (\mathit{bimap}\ \langle \alpha \rangle)\ a)
\end{array}
$$

$$
\begin{array}{lll}
 & & \mathit{Inr}\ b \to \mathit{Inr}\ (\mathit{from}\ (\mathit{bimap}\ \langle\beta\rangle)\ b) \\
 & \mathit{toSum}\ x & = \mathbf{case}\ x\ \mathbf{of} \\
 & & \mathit{Inl}\ a \to \mathit{Inl}\ (\mathit{to}\ (\mathit{bimap}\ \langle\alpha\rangle)\ a) \\
 & & \mathit{Inr}\ b \to \mathit{Inr}\ (\mathit{to}\ (\mathit{bimap}\ \langle\beta\rangle)\ b) \\
\mathbf{in} & \mathit{Conv}\{\mathit{from} = \mathit{fromSum}, \mathit{to} = \mathit{toSum}\} & \\
\mathit{bimap}\ \langle \mathrm{Prod}\ \alpha\ \beta\rangle & = & \\
\mathbf{let} & \mathit{fromProd}\ x = \mathbf{case}\ x\ \mathbf{of} & \\
 & & a \times b \to \mathit{from}\ (\mathit{bimap}\ \langle\alpha\rangle)\ a \times \mathit{to}\ (\mathit{bimap}\ \langle\beta\rangle)\ b \\
 & \mathit{toProd}\ x & = \mathbf{case}\ x\ \mathbf{of} \\
 & & a \times b \to \mathit{to}\ (\mathit{bimap}\ \langle\alpha\rangle)\ a \times \mathit{to}\ (\mathit{bimap}\ \langle\beta\rangle)\ b \\
\mathbf{in} & \mathit{Conv}\{\mathit{from} = \mathit{fromProd}, \mathit{to} = \mathit{toProd}\} & \\
\mathit{bimap}\ \langle \mathrm{Integer}\rangle & = \mathit{Conv}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} & \\
\mathit{bimap}\ \langle \mathrm{Char}\rangle & = \mathit{Conv}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} & \\
\mathit{bimap}\ \langle \mathrm{Float}\rangle & = \mathit{Conv}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} & \\
\mathit{bimap}\ \langle \alpha \to \beta\rangle & = & \\
\mathbf{let} & \mathit{fromFun}\ f & = \mathit{from}\ (\mathit{bimap}\ \langle\beta\rangle) \cdot f \cdot \mathit{to}\ (\mathit{bimap}\ \langle\alpha\rangle) \\
 & \mathit{toFun}\ f & = \mathit{to}\ (\mathit{bimap}\ \langle\beta\rangle) \cdot f \cdot \mathit{from}\ (\mathit{bimap}\ \langle\alpha\rangle) \\
\mathbf{in} & \mathit{Conv}\{\mathit{from} = \mathit{fromFun}, \mathit{to} = \mathit{toFun}\}. &
\end{array}
$$

If we want to use the generic *bimap* function to generate conversions for the fixed-point view, we need to give it a special status within our framework. Actually, in Generic Haskell, it already posesses such a status, since bidirectional mappings play a crucial rôle within the specialization of generic functions [35, 92, 39, 58].

Higher-order Kinded Types

Even with the *bimap* function available for the generation of conversion functions, there is still a class of types that troubles the implementation of the fixed-point view.

For the type of generalized rose trees, for example,

$$\mathbf{data}\ \mathrm{GRose}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star) = \mathit{GBranch}\ \mathrm{a}\ (\mathrm{f}\ (\mathrm{GRose}\ \mathrm{f}\ \mathrm{a})),$$

we derive

$$
\begin{array}{l}
\mathbf{data}\ \mathrm{GRose}^{\bullet}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ (\mathrm{r} :: \star) = \mathit{GBranch}^{\bullet}\ \mathrm{a}\ (\mathrm{f}\ \mathrm{r}) \\
\mathbf{type}\ \mathrm{GRose}^{\circ}_{\mathcal{F}}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star) \qquad\quad = \mathrm{Fix}\ (\mathit{GBranch}^{\bullet}\ \mathrm{f}\ \mathrm{a}).
\end{array}
$$

GRose has kind $(\star \to \star) \to \star \to \star$, i.e., the type parameter f denotes a parametric type rather that a proper type. Consequently, to define a pair of conversion functions,

$$
\begin{array}{l}
\mathit{fromGRose}_{\mathcal{F}} :: \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .\ \mathrm{GRose}\ \mathrm{f}\ \mathrm{a} \to \mathrm{GRose}^{\circ}_{\mathcal{F}}\ \mathrm{f}\ \mathrm{a} \\
\mathit{toGRose}_{\mathcal{F}} \quad :: \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .\ \mathrm{GRose}^{\circ}_{\mathcal{F}}\ \mathrm{f}\ \mathrm{a} \to \mathrm{GRose}\ \mathrm{f}\ \mathrm{a},
\end{array}
$$

we need to be able to map over the type argument f. In Haskell, though, there is no way to define a polymorphic function of type

$$\forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star)\ .\ (\mathrm{a} \to \mathrm{b}) \to \mathrm{f}\ \mathrm{a} \to \mathrm{f}\ \mathrm{b}$$

that performs the required mapping.

$$\boxed{[\![D_1]\!]^{\mathsf{ptr}} \equiv D_2}$$

$$\frac{\begin{array}{c}\{a_{\ell+1} \not\equiv a_i\}^{i\in 1..\ell} \\ \{\{t'_{j,k} \equiv [a_{\ell+1} \,/\, T\ \{a_i\}^{i\in 1..\ell}]\ t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}\end{array}}{\begin{array}{l}[\![\mathbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\mathsf{ptr}} \\ \quad \equiv \mathbf{data}\ \mathsf{ptr}(T) = \\ \qquad \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \Lambda a_{\ell+1} :: \star\ .\ \{\mathsf{ptr}(C_j)\ \{t'_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}\end{array}} \quad \text{(ptr-data)}$$

Figure 7.1: Pattern functors

One solution may be to allow conversion functions to rely on generic functions. Then we can ensure that suitable mapping functions are available for all type arguments by letting the conversion functions for the fixed-point view depend on the generic *bimap* functions. (Note that this approach is much more demanding than the situation that occurred in the previous subsection, where we proposed to use *bimap* at compile time to derive mappings for known data types. Here, we require *bimap* instances for unknown type arguments to be passed around at run time.) However, having conversions rely on *bimap*, introduces additional dependencies for all generic functions and requires the underlying theory of Dependency-style Generic Haskell to be adapted. Furthermore, the additional dependencies make that the size of the generated Haskell code increases dramatically. Although it is expected that this apparent source of inefficiency is subject to optimization [87, 2, 3], a thorough study of this approach remains part of our future work.

Here, we adopt a pragmatic solution and exclude higher-order kinded data types from the view domain of $\mathcal{F}$. In the next section, an alternative view is presented that does not suffer from these problems with kinds of higher order. In this section, we proceed by discussing the formalities that involve the generation of structure types and conversions for the fixed-point view.

Pattern Functors

An essential aspect of the fixed-point view is the automatic derivation of pattern functors.

Given a declaration D_1 of type T, a declaration D_2 for the pattern functor $\mathsf{ptr}(T)$ is generated according the rule (ptr-data), which is shown in Figure 7.1 and takes the form

$$[\![D_1]\!]^{\mathsf{ptr}} \equiv D_2.$$

The metafunction ptr is assumed to produce a unique name for the functor. The definition of $\mathsf{ptr}(T)$ follows the structure of T, replacing all recursive calls by an extra type argument.

$$\boxed{\mathsf{viewtypes}_{\mathcal{F}} \equiv \mathrm{K}; \Gamma}$$

$$\frac{}{\mathsf{viewtypes}_{\mathcal{F}} \equiv \mathrm{Fix} :: (\star \to \star) \to \star; \mathit{In} :: \forall \mathrm{f} :: \star \to \star \,.\, \mathrm{f}\ (\mathrm{Fix}\ \mathrm{f}) \to \mathrm{Fix}\ \mathrm{f}} \quad \text{(vt-fix)}$$

Figure 7.2: View types for the fixed-point view

$$\boxed{\mathcal{F}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n}}$$

$$\frac{D \equiv \mathbf{data}\ T = \{\Lambda a_i :: \star \,.\,\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}}{\mathcal{F}[\![D]\!]^{\mathrm{str}} \equiv \{\Lambda a_i :: \star \,.\,\}^{i \in 1..\ell}\ \mathrm{Fix}\ (\mathsf{ptr}(T)\ \{a_i\}^{i \in 1..\ell}); [\![D]\!]^{\mathrm{ptr}}} \quad \text{(str-fix-data)}$$

Figure 7.3: Structural representation of data types in the fixed-point view

View Types

The sole view type of the fixed-point view is Fix:

$$\mathbf{data}\ \mathrm{Fix} = \Lambda \mathrm{f} :: \star \to \star \,.\, \mathit{In}\ (\mathrm{f}\ (\mathrm{Fix}\ \mathrm{f})).$$

The bindings for Fix are shown in Figure 7.2. Fix has a second-order kind, $(\star \to \star) \to \star$; the data constructor *In* has type $\forall \mathrm{f} :: \star \to \star \,.\, \mathrm{f}\ (\mathrm{Fix}\ \mathrm{f}) \to \mathrm{Fix}\ \mathrm{f}$.

Generating Structure Types

The rule for generating structure types for $\mathcal{F}$, (str-fix-data), is straightforward. It is depicted in Figure 7.3 and has the form

$$\mathcal{F}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n}.$$

The rule simply states that a structure type is derived by applying the type Fix to a partially applied pattern functor. The declaration of the pattern functor is emitted as a supporting declaration. Note that the parameters of the original data type are restricted to kind $\star$, effectively excluding higher-order kinded types from the view domain.

Generating Conversions

Generating conversion functions for $\mathcal{F}$ is more involved. The algorithm is presented in Figures 7.4 and 7.5. It consists of judgements of the forms

$$\begin{array}{ll} \mathcal{F}[\![D_0]\!]^{\mathrm{conv}} & \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \\ \mathcal{F}[\![\{t_k\}^{k \in 1..n}]\!]^{\mathrm{conv}}_{t_0;e,e'} & \equiv \{p_k\}^{k \in 1..n}; \{e_k\}^{k \in 1..n}. \end{array}$$

The first form indicates that conversion functions e_{from} and e_{to} are derived based on the structure of a type declaration D_0. The second form expresses

$$\boxed{\mathcal{F}[\![D]\!]^{\mathrm{conv}} \equiv e_{\mathrm{from}}; e_{\mathrm{to}}}$$

$$\frac{\begin{array}{c} t_0 \equiv T\ \{a_i\}^{i\in 1..\ell} \\ \{\mathcal{F}[\![\{t_{j,k}\}^{k\in 1..n_j}]\!]^{\mathrm{conv}}_{t_0;x_{\mathrm{from}};x_{\mathrm{to}}} \equiv \{p_{j,k}\}^{k\in 1..n_j}; \{e_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m} \\ \{\mathcal{F}[\![\{t_{j,k}\}^{k\in 1..n_j}]\!]^{\mathrm{conv}}_{t_0;x_{\mathrm{to}};x_{\mathrm{from}}} \equiv \{p'_{j,k}\}^{k\in 1..n_j}; \{e'_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m} \\ x_{\mathrm{from}} \notin \{\{\mathsf{pv}(p_{j,k}), \mathsf{pv}(p'_{j,k})\}^{k\in 1..n_j}_{,}\}^{j\in 1..m}_{,} \\ x_{\mathrm{to}} \notin \{\{\mathsf{pv}(p_{j,k}), \mathsf{pv}(p'_{j,k}),\}^{k\in 1..n_j}\}^{j\in 1..m}\ x_{\mathrm{from}} \\ d_{\mathrm{from}} \equiv x_{\mathrm{from}} = \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of} \\ \quad \{C_j\ \{p_{j,k}\}^{k\in 1..n_j} \to In\ (\mathsf{ptr}(C_j)\ \{e_{j,k}\}^{k\in 1..n_j})\}^{j\in 1..m} \\ d_{\mathrm{to}} \equiv x_{\mathrm{to}} = \lambda x\ .\ \mathbf{case}\ x\ \mathbf{of} \\ \quad \{In\ (\mathsf{ptr}(C_j)\ \{p'_{j,k}\}^{k\in 1..n_j}) \to C_j\ \{e'_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m} \\ e_{\mathrm{from}} \equiv \mathbf{let}\ d_{\mathrm{from}}; d_{\mathrm{to}}\ \mathbf{in}\ x_{\mathrm{from}} \qquad e_{\mathrm{to}} \equiv \mathbf{let}\ d_{\mathrm{from}}; d_{\mathrm{to}}\ \mathbf{in}\ x_{\mathrm{to}} \end{array}}{\begin{array}{r} \mathcal{F}[\![\mathbf{data}\ T = \{\Lambda a_i :: \star\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m}_{|}]\!]^{\mathrm{conv}} \\ \equiv e_{\mathrm{from}}; e_{\mathrm{to}} \end{array}} \quad \text{(conv-fix-data)}$$

Figure 7.4: Conversions for data types in the fixed-point view

$$\boxed{\mathcal{F}[\![\{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}}_{t_0;e;e'} \equiv \{p_k\}^{k\in 1..n}; \{e_k\}^{k\in 1..n}}$$

$$\frac{}{\mathcal{F}[\![\varepsilon]\!]^{\mathrm{conv}}_{t_0;e_{\mathrm{conv}};e'_{\mathrm{conv}}} \equiv \varepsilon;\varepsilon} \quad \text{(conv-fix-fld-1)}$$

$$\frac{\begin{array}{c} n \in 1\,..\qquad t_1 \equiv t_0 \qquad \{x_1\ \not\equiv x_k\}^{k\in 2..n} \\ \mathcal{F}[\![\{t_k\}^{k\in 2..n}]\!]^{\mathrm{conv}}_{t_0;e_{\mathrm{conv}};e'_{\mathrm{conv}}} \equiv \{x_k\}^{k\in 2..n}; \{e_k\}^{k\in 2..n} \end{array}}{\mathcal{F}[\![\{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}}_{t_0;e_{\mathrm{conv}};e'_{\mathrm{conv}}} \equiv \{x_k\}^{k\in 1..n}; (e_{\mathrm{conv}}\ x_1)\ \{e_k\}^{k\in 2..n}} \quad \text{(conv-fix-fld-2)}$$

$$\frac{\begin{array}{c} n \in 1\,..\qquad t_1 \not\equiv t_0 \qquad \{x_1\ \not\equiv x_k\}^{k\in 2..n} \\ \mathcal{F}[\![\{t_k\}^{k\in 2..n}]\!]^{\mathrm{conv}}_{t_0;e_{\mathrm{conv}};e'_{\mathrm{conv}}} \equiv \{x_k\}^{k\in 2..n}; \{e_k\}^{k\in 2..n} \end{array}}{\begin{array}{r} \mathcal{F}[\![\{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}}_{t_0;e_{\mathrm{conv}};e'_{\mathrm{conv}}} \\ \equiv \{x_k\}^{k\in 1..n}; \mathsf{map}(x_1, t_1, t_0, e_{\mathrm{conv}}, e'_{\mathrm{conv}})\ \{e_k\}^{k\in 2..n} \end{array}} \quad \text{(conv-fix-fld-3)}$$

Figure 7.5: Conversions for fields in the fixed-point view

the generation of pattern-expression pairs $\{p_k\}^{k\in 1..n}; \{e_k\}^{k\in 1..n}$ for a list of constructor fields $\{t_k\}^{k\in 1..n}$; the generation of these pairs is driven by the original type t_0 and the conversion functions e and e'.

Because the conversion functions may be mutually recursive, rule (conv-fix-data) makes use of the core language's recursive **let** construct. To this end, the rule introduces fresh variables x_{from} and x_{to} that refer to the conversions. It relies on the rules for constructor fields to issue the recursive calls to x_{from} and x_{to} in the appropriate positions.

For the generation of patterns and expressions from lists of fields, we distinguish three cases. If the field list is empty, rule (conv-fix-fld-1) applies and no patterns or expressions are generated. If the field list is non-empty, we recursively generate patterns and expressions for its tail, while focussing on the head element. If this head element equals the original type, rule (conv-fix-fld-2) makes sure that the conversion function is applied. Rule (conv-fix-fld-3) deals with the situation in which the head element does not equal the original type. In that case it may be necessary to map the conversion functions over a fixed data structure; here, we assume that we have a metafunction map that takes care of the details.

7.2 Recursive View

In this section, we present a alternative view on data types that also gives access to recursive calls in type declarations. Although, for the generic programmer, this view may appear as slightly more complicated than the fixed-point view, its implementation is in fact better manageable. Moreover, unlike the fixed-point view, this view poses no problems when applied to types of higher-order kinds. To reflect the fact that, on regular recursive types, it is more applicable than the fixed-point view, we dub this view the *recursive view*.

Embedded Conversions

In the recursive view $\mathcal{R}$, recursive calls in data types are modelled by the type Rec. This type is similar to Fix, but additionally embeds a conversion between the original data type and its representation as a fixed point:

$$\textbf{data}\ \text{Rec}\ (\text{f} :: \star \to \star)\ (\text{r} :: \star) = \mathit{Rec}\ (\text{f r})\ (\text{Conv r}\ (\text{Rec f r})).$$

Like Fix, Rec takes a type argument of kind $\star \to \star$, which is used to pass in the pattern functor. Additionally, Rec takes an argument of kind $\star$, that represents the original data type itself.

For instance, in the recursive view, the structural representation of the type Term is given by

$$\textbf{type}\ \text{Term}^{\circ}_{\mathcal{R}} = \text{Rec Term}^{\bullet}\ \text{Term}.$$

A value of type $\text{Term}^{\circ}_{\mathcal{R}}$ consists of two parts: a value of type $\text{Term}^{\bullet}$ Term and a pair of conversion functions between Term and $\text{Term}^{\circ}_{\mathcal{R}}$. The conversion functions are defined as follows:

$$\mathit{fromTerm}_{\mathcal{R}} \qquad :: \text{Term} \to \text{Term}^{\circ}_{\mathcal{R}}$$

$$
\begin{aligned}
&\mathit{fromTerm}_{\mathcal{R}}\ (\mathit{Var}\ x) &&= \mathit{Rec}\ (\mathit{Var}^{\bullet}\ x)\ \mathit{convTerm}_{\mathcal{R}}\\
&\mathit{fromTerm}_{\mathcal{R}}\ (\mathit{Abs}\ x\ t) &&= \mathit{Rec}\ (\mathit{Abs}^{\bullet}\ x\ t)\ \mathit{convTerm}_{\mathcal{R}}\\
&\mathit{fromTerm}_{\mathcal{R}}\ (\mathit{App}\ t_1\ t_2) &&= \mathit{Rec}\ (\mathit{App}^{\bullet}\ t_1\ t_2)\ \mathit{convTerm}_{\mathcal{R}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{toTerm}_{\mathcal{R}} &&:: \mathrm{Term}^{\circ}_{\mathcal{R}} \to \mathrm{Term}\\
&\mathit{toTerm}_{\mathcal{R}}\ (\mathit{Rec}\ (\mathit{Var}^{\bullet}\ x)\ \_) &&= \mathit{Var}\ x\\
&\mathit{toTerm}_{\mathcal{R}}\ (\mathit{Rec}\ (\mathit{Abs}^{\bullet}\ x\ t)\ \_) &&= \mathit{Abs}\ x\ t\\
&\mathit{toTerm}_{\mathcal{R}}\ (\mathit{Rec}\ (\mathit{App}^{\bullet}\ t_1\ t_2)\ \_) &&= \mathit{App}\ t_1\ t_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{convTerm}_{\mathcal{R}} &&:: \mathrm{Conv\ Term\ Term}^{\circ}_{\mathcal{R}}\\
&\mathit{convTerm}_{\mathcal{R}} &&=\\
&\quad \mathit{Conv}\{\mathit{from} = \mathit{fromTerm}_{\mathcal{R}}, \mathit{to} = \mathit{toTerm}_{\mathcal{R}}\}.
\end{aligned}
$$

Instead of applying the conversion functions recursively, they are embedded in the structure values. Hence, we do not need bidirectional mappings to convert recursive values that are contained in additional data structures:

$$
\begin{aligned}
&\mathbf{type}\ \mathrm{Rose}^{\circ}_{\mathcal{R}}\ (\mathrm{a} :: \star) &&= \mathrm{Rec}\ (\mathrm{Rose}^{\bullet}\ \mathrm{a})\ (\mathrm{Rose\ a})\\
&\mathit{fromRose}_{\mathcal{R}} &&:: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Rose\ a} \to \mathrm{Rose}^{\circ}_{\mathcal{R}}\ \mathrm{a}\\
&\mathit{fromRose}_{\mathcal{R}}\ (\mathit{Branch}\ a\ as) &&= \mathit{Rec}\ (\mathit{Branch}^{\bullet}\ a\ as)\ \mathit{convRose}_{\mathcal{R}}\\
&\mathit{toRose}_{\mathcal{R}} &&:: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Rose}^{\circ}_{\mathcal{R}}\ \mathrm{a} \to \mathrm{Rose\ a}\\
&\mathit{toRose}_{\mathcal{R}}\ (\mathit{Rec}\ (\mathit{Branch}^{\bullet}\ a\ as)\ \_) &&= \mathit{Branch}\ a\ as\\
&\mathit{convRose}_{\mathcal{R}} &&:: \forall(\mathrm{a} :: \star)\ .\ \mathrm{Conv\ (Rose\ a)\ (Rose}^{\circ}_{\mathcal{R}}\ \mathrm{a)}\\
&\mathit{convRose}_{\mathcal{R}} &&=\\
&\quad \mathit{Conv}\{\mathit{from} = \mathit{fromRose}_{\mathcal{R}}, \mathit{to} = \mathit{toRose}_{\mathcal{R}}\}.
\end{aligned}
$$

Moreover, we do not encounter problems with higher-order kinded types, as we did for representations involving Fix:

$$
\begin{aligned}
&\mathbf{type}\ \mathrm{GRose}^{\circ}_{\mathcal{R}}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\\
&\qquad\qquad = \mathrm{Rec\ (GRose}^{\bullet}\ \mathrm{f\ a)\ (GRose\ f\ a)}\\
&\mathit{fromGRose}_{\mathcal{R}} ::\\
&\quad \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .\ \mathrm{GRose\ f\ a} \to \mathrm{GRose}^{\circ}_{\mathcal{R}}\ \mathrm{f\ a}\\
&\mathit{fromGRose}_{\mathcal{R}}\ (\mathit{GBranch}\ a\ as) = \mathit{Rec}\ (\mathit{GBranch}^{\bullet}\ a\ as)\ \mathit{convGRose}_{\mathcal{R}}\\
&\mathit{toGRose}_{\mathcal{R}} ::\\
&\quad \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .\ \mathrm{GRose}^{\circ}_{\mathcal{R}}\ \mathrm{f\ a} \to \mathrm{GRose\ f\ a}\\
&\mathit{toGRose}_{\mathcal{R}}\ (\mathit{Rec}\ (\mathit{GBranch}^{\bullet}\ a\ as)\ \_)\\
&\qquad\qquad = \mathit{GBranch}\ a\ as\\
&\mathit{convGRose}_{\mathcal{R}} ::\\
&\quad \forall(\mathrm{f} :: \star \to \star)\ (\mathrm{a} :: \star)\ .\ \mathrm{Conv\ (GRose\ f\ a)\ (GRose}^{\circ}_{\mathcal{R}}\ \mathrm{f\ a)}\\
&\mathit{convGRose}_{\mathcal{R}} =\\
&\quad \mathit{Conv}\{\mathit{from} = \mathit{fromGRose}_{\mathcal{R}}, \mathit{to} = \mathit{toGRose}_{\mathcal{R}}\}.
\end{aligned}
$$

Compared to the fixed-point view, the recursive view gives rise to a slightly more complicated definition of the generic *children* function:

$$
\begin{aligned}
&\mathit{children}\ \langle \mathrm{a} :: \star \leftarrow \mathcal{R} \rangle &&:: (\mathit{collect}\ \langle \mathrm{a} \mid \mathrm{a} \rangle) \Rightarrow \mathrm{a} \to [\mathrm{a}]\\
&\mathit{children}\ \langle \mathrm{Rec}\ \varphi\ \rho \rangle\ (\mathit{Rec}\ r\ \mathit{conv}) &&= \mathbf{let}\ \mathit{collect}\ \langle \alpha \rangle\ a = [\mathit{from}\ \mathit{conv}\ a]\\
&\qquad\qquad\qquad\qquad\qquad\qquad &&\quad \mathbf{in}\ \ \mathit{collect}\ \langle \varphi\ \alpha \rangle\ r.
\end{aligned}
$$

$$\boxed{\mathsf{viewtypes}_{\mathcal{R}} \equiv \mathrm{K}; \Gamma}$$

$$\frac{}{\mathsf{viewtypes}_{\mathcal{R}} \equiv \quad \begin{array}{l} \mathrm{Conv} :: \star \to \star \to \star, \\ \mathrm{Rec} \quad :: (\star \to \star) \to \star \to \star; \\ \mathit{Conv} :: \forall \mathrm{a} :: \star \,.\, \forall \mathrm{b} :: \star \,.\, (\mathrm{a} \to \mathrm{b}) \to (\mathrm{b} \to \mathrm{a}) \to \mathrm{Conv\ a\ b}, \\ \mathit{Rec} \quad :: \forall \mathrm{f} :: \star \to \star \,.\, \forall \mathrm{r} :: \star \,.\, \mathrm{f\ r} \to \mathrm{Conv\ r\ (Rec\ f\ r)} \to \mathrm{Rec\ f\ r} \end{array}} \quad \text{(vt-rec)}$$

Figure 7.6: View types for the recursive view

$$\boxed{\mathcal{R}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n}}$$

$$\frac{D \equiv \mathbf{data}\ T = \{\Lambda a_i :: \kappa_i \,.\,\}^{i \in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k \in 1..n_j}\}_{|}^{j \in 1..m}}{\begin{array}{l} \mathcal{F}[\![D]\!]^{\mathrm{str}} \equiv\ \{\Lambda a_i :: \kappa_i \,.\,\}^{i \in 1..\ell} \\ \qquad\qquad \mathrm{Rec}\ (\mathsf{ptr}(T)\ \{a_i\}^{i \in 1..\ell})\ (T\ \{a_i\}^{i \in 1..\ell}); \\ \qquad [\![D]\!]^{\mathrm{ptr}} \end{array}} \quad \text{(str-rec-data)}$$

Figure 7.7: Structural representation of data types in the recursive view

Notice how we use the embedded conversion to ensure that the list elements have the correct type.

We conclude the section with a formalization of the recursive view.

View Types

The view types of $\mathcal{R}$ are Rec and Conv,

$$\begin{array}{l} \mathbf{data}\ \mathrm{Rec} \quad = \Lambda \mathrm{f} :: \star \to \star \,.\, \Lambda \mathrm{r} :: \star \,.\, \mathit{Rec}\ (\mathrm{f\ r})\ (\mathrm{Conv\ r\ (Rec\ f\ r)}) \\ \mathbf{data}\ \mathrm{Conv} = \Lambda \mathrm{a} :: \star \,.\, \Lambda \mathrm{b} :: \star \,.\, \mathit{Conv}\ (\mathrm{a} \to \mathrm{b})\ (\mathrm{b} \to \mathrm{a}). \end{array}$$

The required bindings are displayed in Figure 7.6.

Generating Structure Types

Structure types for the recursive view are generated by the rule (str-rec-data), which applies the type constructor Rec to the partially applied pattern functor and to the original data type. As a supporting type declaration, the definition of the pattern functor is emitted. The judgement is depicted in Figure 7.7 and takes the form

$$\mathcal{R}[\![D_0]\!]^{\mathrm{str}} \equiv u; \{D_i\}_{,}^{i \in 1..n}.$$

$$\boxed{\mathcal{R}[\![D]\!]^{\text{conv}} \equiv e_{\text{from}}; e_{\text{to}}}$$

$$\frac{\begin{array}{c}\{\mathcal{R}[\![\{t_{j,k}\}^{k\in 1..n_j}]\!]^{\text{conv}} \equiv \{p_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m} \\ x_{\text{from}} \notin \{\{\mathsf{pv}(p_{j,k})\}_{,}^{k\in 1..n_j}\}_{,}^{j\in 1..m} \\ x_{\text{to}} \notin \{\{\mathsf{pv}(p_{j,k}),\}^{k\in 1..n_j}\}^{j\in 1..m}\ x_{\text{from}} \\ x_{\text{conv}} \notin \{\{\mathsf{pv}(p_{j,k}),\}^{k\in 1..n_j}\}^{j\in 1..m}\ x_{\text{from}}, x_{\text{to}} \\ d_{\text{from}} \equiv \\ \quad x_{\text{from}} = \lambda x\ .\ \textbf{case}\ x\ \textbf{of} \\ \quad\quad \{C_j\ \{p_{j,k}\}^{k\in 1..n_j} \to \mathit{Rec}\ (\mathsf{ptr}(C_j)\ \{p_{j,k}\}^{k\in 1..n_j})\ e_{\text{conv}}\}^{j\in 1..m} \\ d_{\text{to}} \equiv \\ \quad x_{\text{to}} = \lambda x\ .\ \textbf{case}\ x\ \textbf{of} \\ \quad\quad \{\mathit{Rec}\ (\mathsf{ptr}(C_j)\ \{p_{j,k}\}^{k\in 1..n_j})\ x_{\text{conv}} \to C_j\ \{p_{j,k}\}^{k\in 1..n_j}\}^{j\in 1..m} \\ \quad\quad\quad e_{\text{conv}} \equiv \mathit{Conv}\ x_{\text{from}}\ x_{\text{to}} \\ e_{\text{from}} \equiv \textbf{let}\ d_{\text{from}}; d_{\text{to}}\ \textbf{in}\ x_{\text{from}} \qquad e_{\text{to}} \equiv \textbf{let}\ d_{\text{from}}; d_{\text{to}}\ \textbf{in}\ x_{\text{to}}\end{array}}{\begin{array}{l}\mathcal{R}[\![\textbf{data}\ T = \{\Lambda a_i :: \kappa_i\ .\ \}^{i\in 1..\ell}\ \{C_j\ \{t_{j,k}\}^{k\in 1..n_j}\}_{|}^{j\in 1..m}]\!]^{\text{conv}} \\ \qquad\qquad \equiv e_{\text{from}}; e_{\text{to}}\end{array}} \quad \text{(conv-rec-data)}$$

Figure 7.8: Conversions for data types in the recursive view

Generating Conversions

The rules that generate the conversion functions are of the forms

$$\begin{array}{ll}\mathcal{R}[\![D_0]\!]^{\text{conv}} & \equiv e_{\text{from}}; e_{\text{to}} \\ \mathcal{R}[\![\{t_k\}^{k\in 1..n}]\!]^{\text{conv}} & \equiv \{p_k\}^{k\in 1..n}.\end{array}$$

They are shown in Figures 7.8 and 7.9.

Because the generated conversion pair is contained within its own definition, rule (conv-rec-data) makes use of the recursive **let** construct. The rule makes sure that the pattern variables that appear in a single arm of a **case** statement are distinct, and places the patterns at the appropriate positions behind the constructors of the original type and the pattern functor.

Were the corresponding rules for the fixed-point view emit additional mapping expressions to make sure that values of the appropriate types are produced and consumed, the rules for constructor fields in the recursive view—(conv-rec-fld-1) and (conv-rec-fld-2)—do nothing more than producing a fresh pattern variable for each field of a constructor.

7.3 Notes

Generic Haskell's precursor, PolyP [46], does also provide access to the recursive calls in the defintion of a data type, and thus enables the definition of a

$$\boxed{\mathcal{R}[\![\{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}} \equiv \{p_k\}^{k\in 1..n}}$$

$$\frac{}{\mathcal{R}[\![\varepsilon]\!]^{\mathrm{conv}} \equiv \varepsilon} \quad \text{(conv-rec-fld-1)}$$

$$\frac{\begin{array}{ll} n \in 1\,.\,. & \{x_1 \not\equiv x_k\}^{k\in 2..n} \\ \mathcal{R}[\![\{t_k\}^{k\in 2..n}]\!]^{\mathrm{conv}} \equiv \{x_k\}^{k\in 2..n} & \end{array}}{\mathcal{R}[\![\{t_k\}^{k\in 1..n}]\!]^{\mathrm{conv}} \equiv \{x_k\}^{k\in 1..n}} \quad \text{(conv-rec-fld-2)}$$

Figure 7.9: Conversions for fields in the recursive view

generic function that collectes the immediate recursive children of a value [48]. Generic functions in PolyP, however, are limited in the sense that they can only be applied to data types of kind $\star \to \star$.

Using the fixed-point view or the recursive view, all functions that can be written in PolyP can also be written in Generic Haskell. In addition, PolyP offers a built-in **FunctorOf** construct to refer to a type-level algebra. This construct can be expressed by type-indexed data types [40, 58] in Generic Haskell.

Chapter 8

Epilogue

We conclude this thesis by sketching some other possible generic views on data types and briefly mentioning some of the implementation issues that arise when an existing Generic Haskell implementation is augmented with views.

8.1 Other Views

Constructors and Labels

The Generic Haskell compiler [17, 18, 60] does not really employ the standard view on data types as presented in this thesis. Instead it uses additional view types to encode information about constructors and record-field labels in a data type [35, 39]. The presence of these view types makes it possible to write generic functions such as *show* and *read* that produce or consume textual representations of values and therefore rely on the names of constructors and labels.

Type Arguments of Higher Kind

The generic functions that were discussed in this thesis were all parameterized by types of kind $\star$. In the literature on generic programming, however, we also encounter generic functions that take type parameters of higher kind. Such functions require alternative view types. Hinze [36], for instance, uses the following types,

$$
\begin{array}{ll}
\textbf{data}\ \mathrm{K}\ (\mathrm{a} :: \star)\ (\mathrm{b} :: \star) & = \mathrm{K}\ \mathrm{a} \\
\textbf{data}\ \mathrm{Id}\ (\mathrm{a} :: \star) & = \mathit{Id}\ \mathrm{a} \\
\textbf{data}\ \mathrm{LSum}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{g} :: \star \to \star)\ (\mathrm{a} :: \star) & = \mathit{LInl}\ (\mathrm{f}\ \mathrm{a}) \mid \mathit{LInr}\ (\mathrm{g}\ \mathrm{a}) \\
\textbf{data}\ \mathrm{LProd}\ (\mathrm{f} :: \star \to \star)\ (\mathrm{g} :: \star \to \star)\ (\mathrm{a} :: \star) & = \mathrm{f}\ \mathrm{a} \otimes \mathrm{g}\ \mathrm{a},
\end{array}
$$

to define *map* as

$$
\begin{array}{ll}
\mathit{map}\ \langle \mathrm{f} :: \star \to \star \rangle & :: \\
\quad (\mathit{map}\ \langle \mathrm{f} \rangle) \Rightarrow \forall (\mathrm{a} :: \star)\ (\mathrm{b} :: \star)\ .\ (\mathrm{a} \to \mathrm{b}) \to \mathrm{f}\ \mathrm{a} \to \mathrm{f}\ \mathrm{b} & \\
\mathit{map}\ \langle \mathrm{K}\ \mathrm{Integer} \rangle\ f\ n & = n \\
\mathit{map}\ \langle \mathrm{K}\ \mathrm{Char} \rangle\ f\ c & = c
\end{array}
$$

$$
\begin{aligned}
&\mathit{map}\ \langle \mathrm{K\ Float} \rangle\ f\ e &&= e \\
&\mathit{map}\ \langle \mathrm{Id} \rangle\ f\ (\mathit{Id}\ a) &&= \mathit{Id}\ (f\ b) \\
&\mathit{map}\ \langle \mathrm{LSum}\ \varphi\ \psi \rangle\ f\ (\mathit{LInl}\ x) &&= \mathit{LInl}\ (\mathit{map}\ \langle \varphi \rangle\ x) \\
&\mathit{map}\ \langle \mathrm{LSum}\ \varphi\ \psi \rangle\ f\ (\mathit{LInr}\ y) &&= \mathit{LInr}\ (\mathit{map}\ \langle \psi \rangle\ y) \\
&\mathit{map}\ \langle \mathrm{LProd}\ \varphi\ \psi \rangle\ f\ (x \otimes y) &&= \mathit{map}\ \langle \varphi \rangle\ x \otimes \mathit{map}\ \langle \psi \rangle\ y.
\end{aligned}
$$

In a later paper by Hinze, Jeuring, and Löh [40], a combination of these type parameters of kind $\star \to \star$ and a fixed-point representation of data types is used to generically implement the Zipper [44]; see also Hinze and Jeuring [38].

Scrap Your Boilerplate

Structural polymorphism is just one approach to functional generic programming. Over the years, several proposals have been made for enriching functional programming languages with generic-programming capabilities. The proposed approaches differ in the amount of expressive power they add to existing languages and the language extensions they require.

Even when they target the same types of applications, it is often hard to transfer ideas and techniques between two different approaches. Therefore, frameworks for generic programming are typically studied in rather isolated environments. When proposals are compared to one another, the focus usually is on issues like which programs can and which programs cannot be written within the context of a specific paradigm: attention is drawn to the surface area of a generic-programming model, rather than to the underlying theory and techniques.

Generic views on data types may allow us to break with this tradition and incorporate different approaches within a single framework. The idea is to define a view that captures the way in which a given approach treates data types and to consequently transfer the building blocks of that approach to Generic Haskell idioms.

One of these other approaches is the *Scrap Your Boilerplate* paradigm [55, 56, 54]. Developed by Ralf Lämmel and Simon Peyton Jones, the Scrap Your Boilerplate approach describes a single pattern for implementing programs that perform traversals over rich, possibly mutually recursive, data structures. Such programs typically consist of large amount of so-called boilerplate code: code that is only there to establish the traversal and that does not contribute to performing the actual task of the program, i.e., the reason for the traversal.

The central steering concepts in the Scrap Your Boilerplate approach are a type-safe cast and a left-biased generic fold operator.

In Generic Haskell type-safe casting can be implemented in terms of a view that labels each structure value with a *type tag* that uniquely identifies the value's original type. Using the same technique that Atanassow and Jeuring [7] apply to generically infer isomorphisms between data types, we can then–at run time–check whether two values share the same type.

Implementing the generic fold in Generic Haskell is a harder nut to crack. It essentially views a value of a data type as either a nullary data constructor or an application of a (partially applied) constructor to an argument value. To capture this approach in a generic view, we can make use of a view type

that has a data constructor for representing constructors and a polymorphic constructor for applications:

$$\begin{aligned}\textbf{data}\ \text{Value}\ (\text{a} :: \star) = {} & \mathit{Con}\ \text{a} \\ & \mid\ \exists(\text{b} :: \star)\ .\ \mathit{ConApp}\ (\text{Value}\ (\text{b} \to \text{a}))\ \text{b}.\end{aligned}$$

Note that the constructor *ConApp* is existentially quantified. Existential types are not part of Haskell 98, but they are supported by GHC. Still, they do not mix very well with generic functions that have dependencies. Fortunately, the generic fold function does not have dependencies.[1] Directly transferring the type of the generic fold to a Generic Haskell type yields

$$\begin{aligned}&\mathit{gfoldl}\ \langle \text{a} :: \star \leftarrow \mathcal{C} \mid \text{f} :: \star \to \star \rangle :: () \Rightarrow \\ &\quad (\forall(\text{c} :: \star)\ (\text{d} :: \star)\ .\ \text{f}\ (\text{a} \to \text{b}) \to \text{a} \to \text{f}\ \text{b}) \\ &\quad \to (\forall(\text{e} :: \star)\ .\ \text{e} \to \text{f}\ \text{e}) \\ &\quad \to \text{a} \\ &\quad \to \text{f}\ \text{a}.\end{aligned}$$

However, applying a non-generic type parameter to a generic type parameter in the type signature of a generic function introduces several non-trivial implementation issues for the specialization algorithm [58]. The same holds for the polymorphic parameters [39]. These issues are not addressed in any current implementation of the Generic Haskell language. Furthermore, to implement *gfoldl* in a view that is based on Value, we require an extra argument—a mapping function for the non-generic parameter f—and an extra data constructor for Value—one that simply wraps a value of the original data type without decomposing it into a constructor and its arguments.

The resulting implementation is quite clumsy; it therefore remains future work to consider alternative views that enable capturing the Scrap Your Boilerplate approach adequately.

Domain-specific Views

The views that are discussed in this thesis share the property that they are applicable to a relatively large class of data types. For instance, the standard, balanced, list-like, and recursive view are applicable to all Haskell data types, and the fixed-point view works for all types with orders less than two.

If one further restricts the class of data types a view should apply to, a multitude of new possibilities arises. If values from specific domains such as SQL [21] or XML [22] are modelled in the Haskell type system, the resulting data types have a special structure that can be exploited by a generic view. For instance, XML Schema [93] has an element 'all' that allows certain elements to appear in any order, whereas in the element 'sequence' the order is fixed. Furthermore, XML Schema allows for mixed content where elements occur interleaved with strings. XML already proved to be an application domain amendable by generic programming [28, 51, 38, 5, 6, 27].

Finite types and **newtype** types are other subclasses of Haskell data types that allow speical treatment.

[1] Löh [58] shows that functions that do not have dependencies—or, more general, that are not reflexive—are problematic in the standard view. For views that employ type tags, however, these issues do not arise.

In the extreme case, a view only works for a single data type, on which it performs an isomorphic transformation. Generic views thus subsume the bidirectional variant of Wadler's views.

8.2 Implementation

A generic view is formally defined by a set of view types and algorithms for generating structure types and conversion functions. To extend a Generic Haskell implementation with the concept of generic views on data types, the syntax of the language needs to be extended with view annotations for type-indexed functions. Furthermore the algorithms for the generation of structural representations and conversions need to be implemented. That way, a fixed amount of alternative views can be added in an ad-hoc fashion.

Towards User-defined Views

Still, it is more desirable to provide a more fundamental extension: one that allows the programmer to define additional views on data types. Therefore he needs to be presented some kind of interface that can be used to specify the appropriate view types and algorithms that make up a user-defined view.

Source-to-source compilers can then use dynamic loading–such as, for example, the Haskell plug-in architecture [73]—to facilitate the addition of new views. The specification of a user-defined view can take the form of a value of a Haskell data type or an instance of a type class. New views do not require to recompile the Generic Haskell compiler, but can be loaded on demand by means of a configuration file or command-line flags.

Template Haskell [84] allows to manipulate abstract syntax trees of Haskell programs and to specify computations that are performed at the compile time of a program. Norell and Jansson [68] describe a way to write a version of Generic Haskell that makes use of Template Haskell. Generic functions then become meta-programs that are specialized at compile time; making use of a Template Haskell library, the need for a separate Generic Haskell compiler disappears. In such a system, views could be implemented as meta-programs as well, and would thus be on the same level with generic functions.

It may even be possible to implement views as generic functions. If we have one ideal, principle view on data types, structure types may be defined as type-indexed types [40] and conversion functions as type-indexed functions. It remains future work to investigate whether this approach is viable.

Bibliography

[1] Serge Abiteboul. On views and XML. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31–June 2, 1999, Philadelphia, Pennsylvania*, pages 1–9. ACM Press, 1999.

[2] Artem Alimarine and Sjaak Smetsers. Optimizing generic functions. In Dexter Kozen, editor, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12–14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 16–31. Springer-Verlag, 2004.

[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 Sympoium, PADL 2005, Long Beach, CA, USA, January 10–11, 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 203–218. Springer-Verlag, 2005.

[4] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11–12, Dagstuhl, Germany*, volume 243 of *IFIP Conference Proceedings*, pages 1–20. Kluwer Academic Publishers, 2003.

[5] Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting xml with generic haskell. In *Proceedings of the 7th Brazilian Symposium on Programming Languages, SBLP 2003*, 2003. The proceedings appeared as a special issue of the Journal of Universal Computer Science.

[6] Frank Atanassow, Dave Clarke, and Johan Jeuring. UUXML: a type-preserving XML Schema-Haskell data binding. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18–19, 2004, Proceedings*, volume 3057 of *Lecture Notes in Computer Science*, pages 71–85. Spinger-Verlag, 2004.

[7] Frank Atanassow and Johan Jeuring. Inferring type isomorphisms generically. In Dexter Kozen, editor, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12–14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 32–53. Springer-Verlag, 2004.

[8] Lennart Augustsson. Cayenne—a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98), Baltimore, Maryland, USA, September 27–29, 1998*, pages 239–250. ACM Press, 1999.

[9] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming—an introduction. In S. Doaitse Swierstra, Pedro Rangel Henriques, and J. Nuno Oliveira, editors, *Advanced Functional Programming, Third International Summer School, Braga, Portugal, September 12–19, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, 1998.

[10] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

[11] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Europe, London, second edition, 1998.

[12] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15–17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.

[13] Rod M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[14] F. Warren Burton and Robert D. Cameron. Pattern mathing with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[15] F. Warren Burton, Erik Meijer, Patrick M. Sansom, Simon Thompson, and Philip Wadler. Views: an extension to Haskell pattern matching, 1996. Proposal. Available from `http://haskell.org/development/views.html`.

[16] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[17] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell's user guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[18] Dave Clarke, Johan Jeuring, and Andres Löh. The Generic Haskell's user guide. Version 1.23 — Beryl release. Technical Report UU-CS-2002-047, Utrecht University, 2002.

[19] Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11–12, Dagstuhl, Germany*, volume 243 of *IFIP Conference Proceedings*, pages 21–47. Kluwer Academic Publishers, 2003.

[20] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, Janaury 1992*, pages 207–212, 1982.

[21] Database language SQL (ISO 9075:1992(e)), 1992. International Organization for Standardization.

[22] Extensible Markup Language (XML) 1.0 (third edition), 2004. World Wide Web Consortium, `http://www.w3c.org/TR/REC-xml-20040204`.

[23] Gerhard Gierz, Karl H. Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael Mislove, and Dana S. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, 1980.

[24] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In Jens Erik Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[25] Jean-Yves Girard. Interprétation fonctionelle et élimination des coupures de l'arithméthique d'ordre supérieur. Thèse d'état, Université Paris VII, 1972.

[26] The Glasgow Haskell Compiler. `http://haskell.org/ghc`.

[27] Rui Guerra, Johan Jeuring, and S. Doaitse Swierstra. Generic validation in an XPath-Haskell data binding, 2005. To appear.

[28] Paul Hagg. A framework for developing generic XML tools. Master's thesis, Utrecht University, 2004.

[29] Robert Harper and J. Gregory Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 23–25, 1995*, pages 130–141. ACM Press, 1995.

[30] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[31] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146(1):29–60, 1969.

[32] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 1999 Haskell Workshop, Friday October 1st, 1999, Paris, France*. Utrecht University, 1999. The proceedings appeared as Technical Report UU-CS-1999-28, Utrecht University, 1999.

[33] Ralf Hinze. Polytypic functions over nested data types. *Discrete Mathematics & Theoretical Computer Science*, 3(4):193–214, 1999.

[34] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, 2000.

[35] Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universität Bonn, 2000.

[36] Ralf Hinze. A new approach to generic functional programming. In *POPL 2000, Proceedings of 27th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages, January 19–21, 2000, Boston, Massachusetts, USA*, pages 119–132. ACM Press, 2000.

[37] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J. Nuno Oliveira, editors, *Mathemathics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3–5, 2000, Proceedings*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, 2000.

[38] Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 57–96. Springer-Verlag, 2003.

[39] Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming, Advanced Lectures*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.

[40] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke A. Boiten and Bernhard Möller, editors, *Mathematics of Program Construction, 6th International Conference, MPC 2002, Dagstuhl Castle, Germany, July 8–10, 2002, Proceedings*, volume 2386 of *Lecture Notes in Computer Science*, pages 148–174. Springer-Verlag, 2002.

[41] Ralf Hinze and Simon Peyton-Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*. Elsevier Science, 2001.

[42] Stefan Holdermans, Johan Jeuring, and Andres Löh. Generic views on data types, 2005. In preparation.

[43] Paul Hudak. *The Haskell School of Expression. Learning Functional Programming Through Multimedia*. Cambridge University Press, Cambridge, 2000.

[44] Gérard P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[45] John Hughes. Restricted data types in Haskell. In Erik Meijer, editor, *Proceedings of the 1999 Haskell Workshop, Friday October 1st, 1999, Paris, France*. Utrecht University, 1999. The proceedings appeared as Technical Report UU-CS-1999-28, Utrecht University, 1999.

[46] Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15–17 January 1997*, pages 470–482. ACM Press, 1997.

[47] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22–28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287. Springer-Verlag, 1999.

[48] Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In Johan Jeuring, editor, *Proceedings Workshop on Generic Programming (WGP2000), July 6, 2000, Ponte de Lima, Portugal*, pages 33–45. Utrecht University, 2000. The proceedings appeared as Technical Report UU-CS-2000-19, Utrecht University, 2000.

[49] C. Barry Jay. Distinguishing data structures and functions: the constructor calculus and functorial types. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2–5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 217–239. Springer-Verlag, 2001.

[50] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, 2004.

[51] Johan Jeuring and Paul Hagg. Generic programming for XML tools. Technical Report UU-CS-2002-023, Utrecht University, 2002.

[52] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24–30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, 1995.

[53] Assaf J. Kfoury and Joe B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 161–174. ACM Press, 1999.

[54] Ralf Lämmel. Modular generic function customisation, 2004. Draft.

[55] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003), New Orleans, LA, USA, January 18, 2003*, pages 26–37. ACM Press, 2003.

[56] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In Chris Okasaki and Kathleen Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19–21, 2004*, pages 244–255. ACM Press, 2004.

[57] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 8(3):157–166, 1966.

[58] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[59] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eight ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25–29, 2003*, pages 141–152. ACM Press, 2003.

[60] Andres Löh, Johan Jeuring, Dave Clarke, Ralf Hinze, Alexey Rodriquez, and Jan de Wit. The Generic Haskell's user guide. Version 1.42 — Coral release. Technical Report UU-CS-2005-004, Utrecht University, 2005.

[61] Grant Malcolm. *Algebraic data types and program transformation*. PhD thesis, University of Groningen, 1990.

[62] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[63] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs, 8th International Symposium, PLILP'96, Aachen, Germany, September 24–27, 1996, Proceedings*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1996.

[64] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Conference Record of FPCA '95 SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture. La Jolle, CA, USA, 25–28 June 1995.*, pages 324–333. ACM Press, 1995.

[65] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[66] John. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.

[67] Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, April 17–19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[68] Ulf Norell and Patrik Jansson. Prototyping generic programming in Template Haskell. In Dexter Kozen, editor, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12–14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 314–333. Springer-Verlag, 2004.

[69] Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24–26, 1994, Minneapolis, Minnesota*, pages 255–266. ACM Press, 1994.

[70] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, 1998.

[71] Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML, Baltimore, Maryland, USA, September, 1998*, pages 14–23, 1998.

[72] Pedro Palao-Gostanza, Ricardo Peña-Marí, and Manuel Núñez. A new look to pattern matching in abstract data types. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96), Philadelphia, Pennsylvania, May 24–26, 1996*, pages 110–121. ACM Press, 1996.

[73] André Pang, Don Stewart, Sean Seefied, and Manuel M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN 2004 Haskell Workshop, September 22, 2004, Snowbird, Utah, USA*, pages 10–21. ACM Press, 2004.

[74] Simon Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems, ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, April 22–24, 1996, Proceedings*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, 1996.

[75] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report*. Cambridge University Press, Cambridge, 2003.

[76] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types, 2004. Under consideration for publication in Journal of Functional Programming.

[77] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2002.

[78] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2005.

[79] François Pottier and Didier Rémy. The essence of ML type inference. In Pierce [78], pages 389–489.

[80] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9–11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

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

[82] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Department of Computer Science, Chalmers University of Technology and Göteborg University, 1998.

[83] Martijn M. Schrage. *Proxima. A presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, 2004.

[84] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[85] Cassio Souza dos Santos, Serge Abiteboul, and Claude Delobel. Virtual schemas and bases. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advanced in Database Technology—EDBT'94, 4th International Conference on Extending Database Techology, Cambridge, United Kingdom, March 28–31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Sciece*, pages 81–94. Springer-Verlag, 1994.

[86] Simon Thompson. Laws in Miranda. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, August 4–6, 1986, Cambridge, Massachusetts, USA*, pages 1–12. ACM Press, 1986.

[87] Martijn de Vries. Specializing type-indexed values by partial evaluation. Master's thesis, University of Groningen, 2004.

[88] Philip Wadler. How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. In Jean-Pierre Jouannaud, editor, *Functional Programminig Languages and Computer Architecture, Nancy, France, September 16–19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1985.

[89] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 1987*, pages 307–313. ACM Press, 1987.

[90] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas*, pages 60–76. ACM Press, 1989.

[91] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 98–114. Springer-Verlag, 2002.

[92] Jan de Wit. A technical overview of Generic Haskell. Master's thesis, Utrecht University, 2002.

[93] XML Schema part 0: primer second edition, 2004. World Wide Web Consortium, `http://www.w3c.org/TR/xmlschema-0`.

Index

$(\circledast)$, 48, 49
$(:)$, 22, 31, 37, 57, 78
$(:)^\bullet$, 80
$(\oplus)$, 48, 49
$(\otimes)$, 91
$(\times)$, 36
$(\rightarrow)$
    kind constructor, 10
    type constructor, 11
$(\curlywedge)$, 48, 49
$(\curlyvee)$, 50
$[\,]$
    data constructor, 22, 31, 37, 57, 78
    type constructor, 22, 31, 37, 57, 78
$[\,]^\bullet$
    data constructor, 80
    type constructor, 80
$[\,]^\circ_{\mathcal{F}}$, 80
$[\,]^\circ$, 37, 57
$\bot$, 9
    run-time failure, 18
$\star$, 10

0, 32, 64
1, 32, 64

Abs$^\bullet$, 80
Abs, 42, 77
AbsBase, 78
abstract data type, 48
abstraction, 9
algebraic data type, 52
App$^\bullet$, 80
App, 42, 77
AppBase, 78
application, 9

bftP, 41
bimap, 81
Bin, 32, 64
Bit, 32, 64
Bool$^\circ$, 57
Bool, 9, 22, 31, 37, 51, 57
Bool$^\circ$, 37, 51
Branch$^\bullet$, 81
Branch, 10, 22, 31, 37, 57, 81

case, 9
Cayenne, 2
children, 79, 80, 87
collect, 41, 79
Con, 93
ConApp, 93
Cons, 10, 22
ConsBase, 79
constructor calculus, 2
Conv, 52, 57, 88
Conv, 52, 57, 88
convBool, 52, 57
convFork, 57
convGRose$_{\mathcal{R}}$, 87
convGRose, 57
convList, 57
convPerfect, 57
convRose$_{\mathcal{R}}$, 87
convRose, 57
convSequ, 57
convTerm$_{\mathcal{F}}$, 81
convTerm$_{\mathcal{R}}$, 86
convTree, 57
core language, 7–22
    environments, 11–12
    expressions, 7–9
    free variables, 12
    kinding, 12–15, 52
    kinds, 10
    operational semantics, 18–22
    pattern matching, 20, 24
    preservation, 22

programs, 7
progress, 22
projection functions, 21
reduction, 18–21
soundness, 22
substitutions, 12
subsumption, 16–17
types, 9–11, 52
typing, 15–17
of patterns, 16
values, 18
weak-head normal form, 20
well-formedness, 11–18
of programs, 17–18
of type declarations, 17–18

data, 9
data abstraction, 47
data compression, 31–35
data constructor, 9
Day, 64
decodes, 33–34, 38
decodesBool, 32
decodesFork, 32
decodesForkBool, 32
default case, 42–43
dependent types, 2

embedding-projection pair, 56
encode, 33–34, 38, 64, 65
encodeBool, 32
encodeFork, 32
encodeForkBool, 32
encodeGRose, 34
encodePerfect, 35
EndS, 37, 57
Epigram, 3
eq, 40
eq, 35–36
eqCI, 41
equality, 35–36
Even, 50
EvenOrOdd, 50
extends, 42

False, 9, 22, 31, 37, 51, 57
Fix, 78, 84
fixed-point operator, 9
flatten, 41
Fork, 9, 22, 31, 37, 57, 71
$\text{Fork}^{\circ}_{\mathcal{L}}$, 71
Fork°, 57
Fork, 9, 22, 31, 37, 57, 71
Fork°, 37
Friday, 64
from, 52, 57
fromBool, 51, 57
fromFork, 57
$\mathit{fromGRose}_{\mathcal{F}}$, 82
$\mathit{fromGRose}_{\mathcal{R}}$, 87
fromGRose, 57
fromList, 57
fromPerfect, 57
$\mathit{fromRose}_{\mathcal{R}}$, 87
fromRose, 57
fromSequ, 57
$\mathit{fromTerm}_{\mathcal{F}}$, 81
$\mathit{fromTerm}_{\mathcal{R}}$, 86
fromTree, 57

GBranch, 82
$\mathit{GBranch}^{\bullet}$, 82
GBranch, 10, 22, 31, 37, 57
generic abstraction, 41
generic algorithm, 31–36
generic application, 39–40
generic function, 36–40
extensions, 40–44
Generic Haskell, 1
Dependency-style, 30, 44
'Generic Haskell, specifically', 44
generic identity function, 44
generic reduction, 41
gfoldl, 93
$\text{GRose}^{\bullet}$, 82
$\text{GRose}^{\circ}_{\mathcal{F}}$, 82
$\text{GRose}^{\circ}_{\mathcal{R}}$, 87
GRose°, 57
GRose, 10, 22, 31, 37, 57, 82
GRose°, 37

Haskell, 1
Haskell 98, 22–24
newtype, 24
pattern matching, 24

Id, 91
Id, 91
implementation, 94
In, 78, 84

induction, 47
Inl, 36
Inr, 36

Java, xv
Just, 28

K, 91
K, 91
kind, 10
    annotation, 10, 22
    order, 10, 35

let, 9
LInl, 91
LInr, 91
List, 10, 22
List′, 79
ListBase, 79
local redefinition, 40
LProd, 91
LSum, 91

main expression, 7
map, 43, 91
mapList, 43
mapping function, 43
mapTree, 43
Maybe, 28
metavariables, 7
mkNat, 49
Monday, 64
Monoid, 27
Monoid, 27
monoid, 27, 41
monoid, 27–29

Nat, 48
Nat, 48
newtype, 93
Nil, 10, 22
NilBase, 79
Node•, 80
Node, 10, 22, 31, 37, 57, 71, 77
NodeBase, 79
Nothing, 28

Odd, 50
OneS, 37, 57
overloading, 2, 25–27

parameterized type, 52
Path, 69
pattern, 9
pattern calculus, 2
pattern functor, 78
Peano
    type constructor, 48
    view, 49
peano, 49
Peano axioms, 48
Perfect°, 57
Perfect, 11, 22, 31, 37, 57
Perfect°, 37
plus, 25–29
plusBool, 25, 26
plusChar, 25, 26
plusInteger, 25, 26
polymorphic recursion, 35
polymorphism
    ad-hoc, 30
    structural, 31
Prod, 36

Rec, 86, 88
Rec, 86, 88
Rose•, 81
Rose°$_{\mathcal{F}}$, 81
Rose°$_{\mathcal{R}}$, 87
Rose°, 57
Rose, 10, 22, 31, 37, 57, 81
Rose°, 37

Saturday, 64
Scrap Your Boilerplate, 92
Sequ°, 57
Sequ, 37, 57
Sequ°, 38
Set, 22
Set, 22
show, 69
showPart, 69
showPart′, 70
smart constructor, 49
specialization error, 27
SQL, 93
structural representation, 38
structure type, 38
structure value, 38
subterms, 77
subtrees, 77

Succ
    data constructor, 48
    view constructor, 49
SuccP, 11, 22, 31, 37, 57
Sum, 36
sums and products, 36–38
Sunday, 64

tails, 78
Term$^{\bullet}$, 80
Term$^{\circ}_{\mathcal{F}}$, 80
Term$^{\circ}_{\mathcal{R}}$, 86
Term, 42, 77
Term$'$, 78
TermBase, 78
Thursday, 64
Tip$^{\bullet}$, 80
Tip, 10, 22, 31, 37, 57, 71, 77
TipBase, 79
to, 52, 57
toBool, 52, 57
toFork, 57
toGRose$_{\mathcal{F}}$, 82
toGRose$_{\mathcal{R}}$, 87
toGRose, 57
toList, 57
toPerfect, 57
toRose$_{\mathcal{R}}$, 87
toRose, 57
toSequ, 57
toTerm$_{\mathcal{F}}$, 81
toTerm$_{\mathcal{R}}$, 86
toTree, 57
Tree$^{\bullet}$, 80
Tree$^{\circ}_{\mathcal{F}}$, 80
Tree$^{\circ}_{\mathcal{L}}$, 71
Tree$^{\circ}$, 57
Tree, 10, 22, 31, 37, 57, 71, 77
Tree$'$, 79
TreeBase, 79
Tree$^{\circ}$, 37
True, 9, 22, 31, 37, 51, 57
Tuesday, 64
tuples, 11
type
    enumeration, 9
    finite, 10
    nested, 11, 35
    rank, 23–24, 35
    record, 9
    recursive, 10
    regular, 11
type application, 9
type class, 2
type constructor, 9
type parameter
    multiple, 43–44
    non-generic, 41
type synonym, 52
type variable, 9
type-index, 25–26
type-indexed function, 25–30
    dependency, 27–29
    dependency constraint, 29
    dependency variable, 28
    reflexive, 29, 35
    signature, 27
    type signature, 26–27, 29

Unit, 36
Unit, 36
universal quantification, 9
Unused, 22

Value, 93
Var$^{\bullet}$, 80
Var, 42, 77
VarBase, 78
varcollect, 42
Variable, 42, 77
Variable, 42, 77
variable, 9
view
    balanced, 64–67
    constructors and labels, 91
    conversion, 51–52
        well-behaved, 55
        well-typed, 55
    domain-specific, 93–94
    fixed-point, 77–86
    generic, 50–55
        definition, 54
    kind preservation, 54
    list-like, 69–74
    recursive, 86–89
    Scrap Your Boilerplate, 92–93
    standard, 51, 57–64
    valid, 55
view annotation, 65
view constructor, 49

view domain, 51
view transformation, 49
view type, 51

Wednesday, 64

XML, 56, 93
XML Schema, 93

Zero, 23
    data constructor, 48
    view constructor, 49
zero, 27–29
ZeroE, 50
ZeroP, 11, 22, 31, 37, 57
ZeroS, 37, 57
Zipper, 92