Ralf Hinze

Generic Programs and Proofs

Bonn, 2000

Für Anja, Lisa und Florian

Brief contents

1 Introduction **1**

2 Background **15**

3 Generic programs **51**

4 Generic proofs **85**

5 Examples **105**

6 Generic Haskell **145**

References **157**

Summary **165**

Curriculum Vitæ **167**

Contents

1 Introduction **1**
1.1 Generic programming in a nutshell . . . 3
1.2 Overview . . . 13

2 Background **15**
2.1 The type system of Haskell . . . 15
2.2 The class system of Haskell . . . 21
2.3 Category theory . . . 24
2.4 The simply typed λ-calculus . . . 31
2.5 The polymorphic λ-calculus . . . 41

3 Generic programs **51**
3.1 Type-indexed values . . . 51
3.2 Generalizing to first- and second-order kinds . . . 66
3.3 Type-indexed values with kind-indexed types . . . 73
3.4 Related work . . . 82

4 Generic proofs **85**
4.1 Fixed point induction . . . 85
4.2 Deriving generic programs . . . 90
4.3 Generic logical relations . . . 94

5 Examples **105**
5.1 Comparison functions . . . 105
5.2 Mapping functions . . . 106
5.3 Zipping functions . . . 113
5.4 Reductions . . . 117
5.5 Generic dictionaries . . . 123
5.6 Generic memo tables . . . 134

6 Generic Haskell **145**
6.1 Implementation . . . 145
6.2 Extensions . . . 154

References **157**

Summary **165**

Curriculum Vitæ **167**

Detailed contents

1 Introduction **1**
- 1.1 Generic programming in a nutshell . . . 3
  - 1.1.1 Binary encoding . . . 3
  - 1.1.2 Size functions . . . 10
- 1.2 Overview . . . 13

2 Background **15**
- 2.1 The type system of Haskell . . . 15
  - 2.1.1 Finite types . . . 16
  - 2.1.2 Regular types . . . 16
  - 2.1.3 Nested types . . . 18
  - 2.1.4 Functional types . . . 20
- 2.2 The class system of Haskell . . . 21
  - 2.2.1 Type classes . . . 21
  - 2.2.2 Constructor classes . . . 22
- 2.3 Category theory . . . 24
  - 2.3.1 Categories, functors and natural transformations . . . 24
  - 2.3.2 Initial objects . . . 25
  - 2.3.3 Terminal objects . . . 25
  - 2.3.4 Products . . . 26
  - 2.3.5 Coproducts . . . 26
  - 2.3.6 Exponentials . . . 27
  - 2.3.7 Isomorphisms . . . 28
  - 2.3.8 Fixed points . . . 29
  - 2.3.9 A semantics for **data** declarations . . . 30
- 2.4 The simply typed λ-calculus . . . 31
  - 2.4.1 Syntax . . . 31
  - 2.4.2 Semantics . . . 33
  - 2.4.3 Böhm trees . . . 35
  - 2.4.4 Logical relations . . . 37
- 2.5 The polymorphic λ-calculus . . . 41
  - 2.5.1 Syntax . . . 42
  - 2.5.2 Semantics . . . 47

3 Generic programs **51**
- 3.1 Type-indexed values . . . 51
  - 3.1.1 Normal forms of types . . . 52
  - 3.1.2 Defining generic values . . . 53
  - 3.1.3 Specializing generic values . . . 57

3.2 Generalizing to first- and second-order kinds 66
3.2.1 Type indices of kind $\star \to \star$ 66
3.2.2 Type indices of kind $(\star \to \star) \to \star \to \star$ 67
3.2.3 Normal forms of types 69
3.2.4 Defining generic values 71
3.2.5 Specializing generic values 71
3.2.6 Limitations of the approach 73
3.3 Type-indexed values with kind-indexed types 73
3.3.1 Defining generic values 76
3.3.2 Specializing generic values 77
3.3.3 Examples 80
3.4 Related work 82

4 Generic proofs **85**
4.1 Fixed point induction 85
4.1.1 Type-indexed values 85
4.1.2 Generalizing to first- and second-order kinds 86
4.2 Deriving generic programs 90
4.3 Generic logical relations 94
4.3.1 Soundness 96
4.3.2 Examples 100

5 Examples **105**
5.1 Comparison functions 105
5.2 Mapping functions 106
5.2.1 Embedding-projection maps 106
5.2.2 Monadic maps 109
5.3 Zipping functions 113
5.4 Reductions 117
5.4.1 POPL-style reductions 117
5.4.2 MPC-style reductions 118
5.4.3 Properties 120
5.4.4 Right and left reductions 121
5.5 Generic dictionaries 123
5.5.1 Introduction 123
5.5.2 Signature 126
5.5.3 Type-indexed tries 126
5.5.4 Empty tries 129
5.5.5 Singleton tries 131
5.5.6 Look up 131
5.5.7 Inserting and merging 132
5.5.8 Properties 134
5.5.9 Related work 134
5.6 Generic memo tables 134
5.6.1 Introduction 134
5.6.2 Signature 135
5.6.3 Memo tables 135
5.6.4 Table look-up 136
5.6.5 Tabulation 137
5.6.6 Properties 141

6 Generic Haskell **145**
6.1 Implementation 145
6.1.1 Generic representation types 146
6.1.2 Specializing generic values 147
6.1.3 Generating embedding-projection maps 151
6.1.4 Encoding rank-n types 152
6.2 Extensions 154
6.2.1 Ad-hoc definitions 154
6.2.2 Constructor names and record labels 155

References **157**

Summary **165**

Curriculum Vitæ **167**

Chapter 1

Introduction

A generic program is one that the programmer writes once, but which works over many different data types. A generic proof is one that the programmer shows once, but which holds for many different data types. This thesis describes a novel approach to functional generic programming and reasoning that is both simpler and more general than previous approaches.

It is widely accepted that type systems are indispensable for building large and reliable software systems. Types provide machine checkable documentation and are often helpful in finding programming errors at an early stage. Polymorphism complements type security by flexibility. Polymorphic type systems like the Hindley-Milner system (Milner 1978) allow the definition of functions that behave uniformly over all types. However, polymorphic type systems are sometimes less flexible that one would wish. For instance, it is not possible to define a polymorphic equality function that works for all types.[1] As a consequence, the programmer is forced to program a separate equality function for each data type from scratch.

Generic, or polytypic, programming (Bird, de Moor, and Hoogendijk 1996; Backhouse, Jansson, Jeuring, and Meertens 1999) addresses this problem. Actually, equality serves as a standard example of a generic function. Further examples are parsing and pretty printing, serialising, ordering, hashing, and so on. Broadly speaking, generic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined data types. A generic function such as a pretty printer or a parser is written once and for all times; its specialization to different instances of data types happens without further effort from the user. This way generic programming greatly simplifies the construction and maintenance of software systems as it automatically adapts functions to changes in the representation of data.

The basic idea of generic programming is to define a function such as taking equality by *induction on the structure of types*. Thus, generic equality takes three arguments, a type and two values of that type, and proceeds by case analysis on the type argument. In other words, generic equality is a function that depends on a type. To put this statement into a broader perspective let us take a look at the structure of a modern functional programming language such as Haskell 98 (Peyton Jones and Hughes 1999). If we ignore the module system, Haskell 98 has the three level structure depicted on the right. The lowest level, that is, the level where the computations take place, consists of *values*. The second level, which imposes structure on the value level, is inhabited by *types*. Finally, on the third level, which imposes structure on the type level, we have so-called *kinds*. Why is there a third level? Now, Haskell allows the programmer to define

kinds
types
values

[1] The parametricity theorem (Wadler 1989) implies that a function of type $\forall A \, . \, A \to A \to \mathit{Bool}$ must necessarily be constant.

parametric types such as the popular data type of lists. The list type constructor can be seen as a function on types and the kind system allows to specify this in a precise way. Thus, a kind is simply the 'type' of a type constructor.

In ordinary programming we routinely define values that depend on values, that is, functions and types that depend on types, that is, type constructors. However, we can also imagine to have dependencies between adjacent levels. For instance, a type might depend on a value or a type might depend on a kind. The following table lists the possible combinations:

kinds depending on kinds	parametric and kind-indexed kinds
kinds depending on types	dependent kinds
types depending on kinds	polymorphic and kind-indexed types
types depending on types	parametric and type-indexed types
types depending on values	dependent types
values depending on types	polymorphic and type-indexed functions
values depending on values	ordinary functions

.

If a higher level depends on a lower level we have so-called dependent types or dependent kinds. Programming languages with dependent types are the subject of intensive research, see, for instance, (Augustsson 1999). Dependent types will, however, play little rôle in this thesis as generic programming is concerned with the opposite direction, where a lower level depends on the same or a higher level. For instance, if a value depends on a type we either have a *polymorphic* or a *type-indexed* function. In both cases the function takes a type as an argument. What is the difference between the two? Now, a polymorphic function stands for an algorithm that happens to be insensitive to what type the values in some structure are. Take, for example, the *length* function that calculates the length of a list. Since it need not inspect the elements of a given list, it has type $\forall A \,.\, List\ A \to Int$. By contrast, a type-indexed function is defined by induction on the structure of its type argument. In some sense, the type argument guides the computation which is performed on the value arguments.

A similar distinction applies to the type and to the kind level: a parametric type does not inspect its type argument whereas a type-indexed type is defined by induction on the structure of its type argument and similarly for kinds. The following table summarizes the interesting cases.

kinds defined by induction on the structure of kinds	kind-indexed kinds
kinds defined by induction on the structure of types	—
types defined by induction on the structure of kinds	kind-indexed types
types defined by induction on the structure of types	type-indexed types
types defined by induction on the structure of values	—
values defined by induction on the structure of types	type-indexed values
values defined by induction on the structure of values	—

We will encounter examples of all sorts of parameterization in this thesis. Of course, the main bulk of the text is concerned with type-indexed functions. Sections 3.1 and 3.2 cover this topic in considerable depth. Perhaps surprisingly, kind-indexed types will also play a prominent rôle since they allow for a more flexible definition of type-indexed functions. This is detailed in Section 3.3. Polytypic types and kind-indexed kinds are less frequent (and also more exotic). They will be dealt with in later sections (Sections 5.5 and 5.6).

The rest of this introduction is structured as follows. Section 1.1 introduces generic functional programming from the programmer's perspective. We will get to know several type-indexed functions and we will see an example of a generic proof. Section 1.2 gives an overview of the remaining chapters.

1.1 Generic programming in a nutshell

Defining a function by induction on the structure of types sounds like a hard nut to crack. We are trained to define functions by induction on the structure of values. Types are used to guide this process, but we typically think of them as separate entities. So, at first sight, generic programming appears to add an extra level of complication and abstraction to programming. However, I claim that generic programming is in many cases actually simpler than conventional programming. The fundamental reason is that genericity gives you 'a lot of things for free'—we will make this statement more precise in the course of this thesis. For the moment, let me support the claim by defining two simple algorithms both in a conventional and in a generic style. Of course, we will consider algorithms that make sense for a large class of data types. Consequently, in the conventional style we have to provide an algorithm for each instance of the class.

REMARK 1.1 The examples in this section and indeed most of the examples in this thesis are given in the functional programming language Haskell 98 (Peyton Jones and Hughes 1999). However, for reasons of coherence we will slightly deviate from Haskell's lexical syntax: both type constructors and type variables are written with an initial upper-case letter (in Haskell type variables begin with a lower-case letter) and both value constructors and value variables are written with an initial lower-case letter (in Haskell value constructors begin with an upper-case letter). This convention helps to easily identify values and types. Furthermore, we write polymorphic types such as $\forall A \,.\, \mathit{List}\ A \to \mathit{Int}$ using an explicit universal quantifier. Unfortunately, in Haskell there is no syntax for universal quantification. □

1.1.1 Binary encoding

The first problem we look at is to encode elements of a given data type as bit streams implementing a simple form of data compression (Jansson and Jeuring 1999). For concreteness, we assume that bit streams are given by the following data type:

$$\begin{aligned} \mathbf{type}\ \mathit{Bin} &= [\mathit{Bit}] \\ \mathbf{data}\ \mathit{Bit} &= \mathtt{0} \mid \mathtt{1}. \end{aligned}$$

Thus, a bit stream is simply a list of bits (see Section 2.1.2 for a short review of Haskell's list syntax). A real implementation might have a more sophisticated representation for Bin but that is a separate matter.

Ad-hoc programs We will implement binary encoders and decoders for three different data types. We consider the types in increasing level of difficulty. The first type defines character strings:

$$\mathbf{data}\ \mathit{String} = \mathit{nilS} \mid \mathit{consS}\ \mathit{Char}\ \mathit{String}.$$

The data type declaration introduces a new type, *String*, and two new value constructors, *nilS* and *consS*. Here is an example element of *String*:

$$consS\ \texttt{'F'}\ (consS\ \texttt{'l'}\ (consS\ \texttt{'o'}\ (consS\ \texttt{'r'}\ (consS\ \texttt{'i'}\ (consS\ \texttt{'a'}\ (consS\ \texttt{'n'}\ nilS)))))).$$

Supposing that $encodeChar :: Char \to Bin$ is an encoder for characters provided from somewhere, we can encode an element of type *String* as follows:

$$\begin{array}{lcl} encodeString & :: & String \to Bin \\ encodeString\ nilS & = & \texttt{0} : [\,] \\ encodeString\ (consS\ c\ s) & = & \texttt{1} : encodeChar\ c +\!\!+\ encodeString\ s. \end{array}$$

We emit one bit to distinguish between the two constructors *nilS* and *consS*. If the argument is a non-empty string of the form $consS\ c\ s$, we (recursively) encode the components c and s and finally concatenate the resulting bit streams.

Given this scheme it is relatively simple to decode a bit stream produced by *encodeString*. Again, we assume that a decoder for characters is provided externally.

$$\begin{array}{lcl} decodesString & :: & Bin \to (String, Bin) \\ decodesString\ [\,] & = & error\ \texttt{"decodesString"} \\ decodesString\ (\texttt{0} : bin) & = & (nilS, bin) \\ decodesString\ (\texttt{1} : bin) & = & \mathbf{let}\ (c, bin_1) = decodesChar\ bin \\ & & \quad\ \ (s, bin_2) = decodesString\ bin_1 \\ & & \mathbf{in}\ (consS\ c\ s, bin_2) \end{array}$$

The decoder has type $Bin \to (String, Bin)$ rather than $Bin \to String$ to be able to compose decoders in a modular fashion: $decodesChar :: Bin \to (Char, Bin)$, for instance, consumes an initial part of the input bit stream and returns the decoded character together with the rest of the input stream. Here are some applications (we assume that characters are encoded in 8 bits).

$$\begin{array}{l} encodeString\ (consS\ \texttt{'L'}\ (consS\ \texttt{'i'}\ (consS\ \texttt{'s'}\ (consS\ \texttt{'a'}\ nilS)))) \\ \Longrightarrow \texttt{1001100101100101101110011101100001100} \\ decodesChar\ (tail\ \texttt{1001100101100101101110011101100001100}) \\ \Longrightarrow (\texttt{'L'}, \texttt{1100101101110011101100001100}) \\ decodesString\ \texttt{1001100101100101101110011101100001100} \\ \Longrightarrow (consS\ \texttt{'L'}\ (consS\ \texttt{'i'}\ (consS\ \texttt{'s'}\ (consS\ \texttt{'a'}\ nilS))), [\,]) \end{array}$$

Note that a string of length n is encoded using $n + 1 + 8 \times n$ bits.

A string is a list of characters. Abstracting over the type of list elements we obtain a more general list type:

$$\mathbf{data}\ List\ A \quad = \quad nil \mid cons\ A\ (List\ A).$$

This parametric type embraces lists of characters of type *List Char*

$$cons\ \texttt{'F'}\ (cons\ \texttt{'l'}\ (cons\ \texttt{'o'}\ (cons\ \texttt{'r'}\ (cons\ \texttt{'i'}\ (cons\ \texttt{'a'}\ (cons\ \texttt{'n'}\ nil)))))),$$

lists of integers of type *List Int*

$$cons\ 2\ (cons\ 3\ (cons\ 5\ (cons\ 7\ (cons\ 11\ (cons\ 13\ nil))))),$$

and so on. Now, how can we encode a list of something? We could insist that the elements of the input list have already been encoded as bit streams. Then *encodeListBin* completes the task:

$$\begin{array}{lcl} encodeListBin & :: & List\ Bin \to Bin \\ encodeListBin\ nil & = & \texttt{0} : [\,] \\ encodeListBin\ (cons\ bin\ bins) & = & \texttt{1} : bin +\!\!+\ encodeListBin\ bins. \end{array}$$

For encoding the elements of a list the following function proves to be useful:

$$
\begin{array}{lcl}
\mathit{mapList} & :: & \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (\mathit{List}\ A_1 \to \mathit{List}\ A_2) \\
\mathit{mapList}\ \mathit{mapA}\ \mathit{nil} & = & \mathit{nil} \\
\mathit{mapList}\ \mathit{mapA}\ (\mathit{cons}\ a\ \mathit{as}) & = & \mathit{cons}\ (\mathit{mapA}\ a)\ (\mathit{mapList}\ \mathit{mapA}\ \mathit{as}).
\end{array}
$$

The function *mapList* is a so-called mapping function that applies a given function to each element of a given list (we will say a lot more about mapping functions in this thesis). Combining *encodeListBin* and *mapList* we can encode a variety of lists:

$$
\begin{array}{l}
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeChar}\ (\mathit{cons}\ \texttt{'A'}\ (\mathit{cons}\ \texttt{'n'}\ (\mathit{cons}\ \texttt{'j'}\ (\mathit{cons}\ \texttt{'a'}\ \mathit{nil})))))\\
\Longrightarrow \texttt{1100000101011101101010101101100001100}\\
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeInt}\ (\mathit{cons}\ 11\ (\mathit{cons}\ 13\ \mathit{nil})))\\
\Longrightarrow \texttt{1110100000000000000000000000000001101100000000000000000000000000000}\\
(\mathit{encodeListBin} \cdot \mathit{mapList}\ (\mathit{encodeListBin} \cdot \mathit{mapList}\ \mathit{encodeBool}))\\
\quad (\mathit{cons}\ (\mathit{cons}\ \mathit{True}\ (\mathit{cons}\ \mathit{False}\ (\mathit{cons}\ \mathit{True}\ \mathit{nil})))\\
\quad (\mathit{cons}\ (\mathit{cons}\ \mathit{False}\ (\mathit{cons}\ \mathit{True}\ (\mathit{cons}\ \mathit{False}\ \mathit{nil})))\\
\quad (\mathit{nil})))\\
\Longrightarrow \texttt{11110110110111000}.
\end{array}
$$

Here, *encodeInt* and *encodeBool* are primitive encoders for integers and Boolean values respectively (an integer occupies 32 bits whereas a Boolean value makes do with one bit).

The million-dollar question is, of course, how do we decode the bit streams thus produced? The first bit tells whether the original list was empty or not, but then we are stuck: we simply do not know how many bits were spent on the first list element. The only way out of this dilemma is to use a decoder function, supplied as an additional argument, that decodes the elements of the original list.

$$
\begin{array}{lcl}
\mathit{decodesList} & :: & \forall A\,.\,(\mathit{Bin} \to (A, \mathit{Bin})) \to (\mathit{Bin} \to (\mathit{List}\ A, \mathit{Bin})) \\
\mathit{decodesList}\ \mathit{decodesA}\ [\,] & = & \mathit{error}\ \texttt{"decodesList"} \\
\mathit{decodesList}\ \mathit{decodesA}\ (\texttt{0} : \mathit{bin}) & = & (\mathit{nil}, \mathit{bin}) \\
\mathit{decodesList}\ \mathit{decodesA}\ (\texttt{1} : \mathit{bin}) & = & \mathbf{let}\ (a, \mathit{bin}_1) = \mathit{decodesA}\ \mathit{bin} \\
 & & \qquad (\mathit{as}, \mathit{bin}_2) = \mathit{decodesList}\ \mathit{decodesA}\ \mathit{bin}_1 \\
 & & \mathbf{in}\ (\mathit{cons}\ a\ \mathit{as}, \mathit{bin}_2)
\end{array}
$$

This definition generalizes *decodeString* defined above; we have $\mathit{decodeString} \cong \mathit{decodesList}\ \mathit{decodesChar}$ (corresponding to $\mathit{String} \cong \mathit{List}\ \mathit{Char}$). In some sense, the abstraction step that led from *String* to *List* is here repeated on the value level. Of course, we can also generalize *encodeString*:

$$
\begin{array}{lcl}
\mathit{encodeList} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{List}\ A \to \mathit{Bin}) \\
\mathit{encodeList}\ \mathit{encodeA}\ \mathit{nil} & = & \texttt{0} : [\,] \\
\mathit{encodeList}\ \mathit{encodeA}\ (\mathit{cons}\ a\ \mathit{as}) & = & \texttt{1} : \mathit{encodeA}\ a \mathbin{+\!\!+} \mathit{encodeList}\ \mathit{encodeA}\ \mathit{as}.
\end{array}
$$

It is not hard to see that $\mathit{encodeList}\ \mathit{encodeA} = \mathit{encodeListBin} \cdot \mathit{mapList}\ \mathit{encodeA}$. Encoding and decoding lists is now fairly simple:

$$\begin{aligned}
&\mathit{encodeList}\ \mathit{encodeChar}\ (\mathit{cons}\ \texttt{'A'}\ (\mathit{cons}\ \texttt{'n'}\ (\mathit{cons}\ \texttt{'j'}\ (\mathit{cons}\ \texttt{'a'}\ \mathit{nil}))))\\
&\Longrightarrow \texttt{1100000101011101101010101101100001100}\\
&\mathit{encodeList}\ \mathit{encodeInt}\ (\mathit{cons}\ 47\ (\mathit{cons}\ 11\ \mathit{nil}))\\
&\Longrightarrow \texttt{111010000000000000000000000000000110110000000000000000000000000000}\\
&\mathit{encodeList}\ (\mathit{encodeList}\ \mathit{encodeBool})\\
&\quad (\mathit{cons}\ (\mathit{cons}\ \mathit{True}\ (\mathit{cons}\ \mathit{False}\ (\mathit{cons}\ \mathit{True}\ \mathit{nil})))\\
&\quad (\mathit{cons}\ (\mathit{cons}\ \mathit{False}\ (\mathit{cons}\ \mathit{True}\ (\mathit{cons}\ \mathit{False}\ \mathit{nil})))\\
&\quad (\mathit{nil})))\\
&\Longrightarrow \texttt{11110110110111000}.
\end{aligned}$$

The third data type we look at provides an alternative to the ubiquitous list type if an efficient indexing operation is required: Okasaki's *binary random-access lists* (1998) support logarithmic access to the elements of a list.

$$\begin{aligned}
\mathbf{data}\ \mathit{Fork}\ A &= \mathit{fork}\ A\ A\\
\mathbf{data}\ \mathit{Sequ}\ A &= \mathit{endS} \mid \mathit{zeroS}\ (\mathit{Sequ}\ (\mathit{Fork}\ A)) \mid \mathit{oneS}\ A\ (\mathit{Sequ}\ (\mathit{Fork}\ A))
\end{aligned}$$

Since the type argument of *Sequ* is changed in the recursive calls, *Sequ* is termed a *nested* or *non-regular* data type (Bird and Meertens 1998). Nested data types are practically important since they can capture data-structural invariants in a way that regular data types cannot. For instance, *Sequ* captures the invariant that binary random-access lists are sequences of perfect binary leaf trees stored in increasing order of height. The following element of type *Sequ Char* illustrates this property:

$$\mathit{oneS}\ \texttt{'F'}\ (\mathit{oneS}\ (\mathit{fork}\ \texttt{'l'}\ \texttt{'o'})\ (\mathit{oneS}\ (\mathit{fork}\ (\mathit{fork}\ \texttt{'r'}\ \texttt{'i'})\ (\mathit{fork}\ \texttt{'a'}\ \texttt{'n'}))\ \mathit{endS})).$$

The first argument of the i-th *oneS* constructor has type $\mathit{Fork}^i\ \mathit{Char}$ ($F^n\ A$ means F applied n times to A), which we may view as the type of perfect binary leaf trees of height i. The sequence above has length 7. Here is a slightly shorter sequence of type *Sequ Int*:

$$\mathit{zeroS}\ (\mathit{oneS}\ (\mathit{fork}\ 2\ 3)\ (\mathit{oneS}\ (\mathit{fork}\ (\mathit{fork}\ 5\ 7)\ (\mathit{fork}\ 11\ 13))\ \mathit{endS})).$$

Note that the constructors *zeroS* and *oneS* encode the length of the list. In other words, the binary representation of the number of elements determines the layout of the binary random-access list. The intimate relationship between the binary number system and this data structure is explained in more detail in (Okasaki 1998; Hinze 2000c), see also Remark 2.2.

Now, using the recursion scheme of *encodeList* we can also program an encoder for binary random-access lists.

$$\begin{aligned}
&\mathit{encodeFork} &&:: \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Fork}\ A \to \mathit{Bin})\\
&\mathit{encodeFork}\ \mathit{encodeA}\ (\mathit{fork}\ a_1\ a_2) &&= \mathit{encodeA}\ a_1 +\!\!\!+\ \mathit{encodeA}\ a_2\\
&\mathit{encodeSequ} &&:: \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Sequ}\ A \to \mathit{Bin})\\
&\mathit{encodeSequ}\ \mathit{encodeA}\ \mathit{endS} &&= \texttt{0} : [\,]\\
&\mathit{encodeSequ}\ \mathit{encodeA}\ (\mathit{zeroS}\ s) &&= \texttt{1} : \texttt{0} : \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{encodeA})\ s\\
&\mathit{encodeSequ}\ \mathit{encodeA}\ (\mathit{oneS}\ a\ s) &&= \texttt{1} : \texttt{1} : \mathit{encodeA}\ a +\!\!\!+\ \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{encodeA})\ s
\end{aligned}$$

Consider the last equation which deals with arguments of the form *oneS a s*. We emit two bits for the constructor and then (recursively) encode its components. Since a has type A, we apply *encodeA*. Similarly, since s has type *Sequ* (*Fork A*), we call *encodeSequ* (*encodeFork encodeA*). The type of the component determines the calls in a straightforward manner. As an aside, note that *encodeSequ* requires a non-schematic form of recursion known as *polymorphic recursion* (Mycroft 1984). The recursive calls are at type $(Fork\ A \to Bin) \to (Sequ\ (Fork\ A) \to Bin)$ which is a substitution instance of the declared type. Functions operating on nested types are in general polymorphically recursive. Haskell 98 allows polymorphic recursion only if an explicit type signature is provided for the function. The rationale behind this restriction is that type inference in the presence of polymorphic recursion is undecidable (Henglein 1993).

$$
\begin{array}{lcl}
decodesFork & :: & \forall A\,.\,(Bin \to (A, Bin)) \to (Bin \to (Fork\ A, Bin)) \\
decodesFork\ decodesA\ bin & = & \mathbf{let}\ (a_1, bin_1) = decodesA\ bin \\
 & & \quad\ \ (a_2, bin_2) = decodesA\ bin_1 \\
 & & \mathbf{in}\ (fork\ a_1\ a_2, bin_2) \\
decodesSequ & :: & \forall A\,.\,(Bin \to (A, Bin)) \to (Bin \to (Sequ\ A, Bin)) \\
decodesSequ\ decodesA\ [\,] & = & error\ \texttt{"decodes"} \\
decodesSequ\ decodesA\ (\texttt{0} : bin) & = & (endS, bin) \\
decodesSequ\ decodesA\ (\texttt{1} : \texttt{0} : bin) & = & \mathbf{let}\ (s, bin') = decodesSequ\ (decodesFork\ decodesA)\ bin \\
 & & \mathbf{in}\ (zeroS\ s, bin') \\
decodesSequ\ decodesA\ (\texttt{1} : \texttt{1} : bin) & = & \mathbf{let}\ (a, bin_1) = decodesA\ bin \\
 & & \quad\ \ (s, bin_2) = decodesSequ\ (decodesFork\ decodesA)\ bin_1 \\
 & & \mathbf{in}\ (oneS\ a\ s, bin_2)
\end{array}
$$

Perhaps surprisingly, encoding a binary random-access list requires less bits than encoding the corresponding list.

$$
\begin{array}{l}
encodeSequ\ encodeChar\ (zeroS\ (zeroS\ (oneS\ (fork\ (fork\ \texttt{'L'}\ \texttt{'i'})\ (fork\ \texttt{'s'}\ \texttt{'a'}))\ endS))) \\
\Longrightarrow \texttt{101011001100101001011011001110100001100} \\
encodeSequ\ encodeInt\ (zeroS\ (oneS\ (fork\ 47\ 11)\ endS)) \\
\Longrightarrow \texttt{1011111101000000000000000000000000000110100000000000000000000000000000}
\end{array}
$$

In general, a string of length n requires $2 \times \lceil lg\ (n+1) \rceil + 1 + 8 \times n$ bits.

Generic programs Before explaining how to define generic versions of *encode* and *decodes* let us first take a closer look at Haskell's data type definitions. The **data** construct combines several features in a single coherent form: type abstraction, type recursion, n-ary sums and n-ary products. The following rewritings of *String*, *List*, *Fork* and *Sequ* make the structure of the data type definitions more explicit.

$$
\begin{array}{lcl}
String & = & 1 + Char \times String \\
List\ A & = & 1 + A \times List\ A \\
Fork\ A & = & A \times A \\
Sequ\ A & = & 1 + Sequ\ (Fork\ A) + A \times Sequ\ (Fork\ A)
\end{array}
$$

Here, '1' denotes the unit type, and '+' and '×' are more conventional notation for binary sums and binary products. For simplicity, we assume that n-ary sums are reduced to binary sums and n-ary products to binary products, that is, read $T_1 + T_2 + T_3$ as $T_1 + (T_2 + T_3)$. In the sequel we treat '1', '+', and '×' as if they were given by the following data type declarations.

$$
\begin{array}{lcl}
\mathbf{data}\ 1 & = & () \\
\mathbf{data}\ A + B & = & inl\ A \mid inr\ B \\
\mathbf{data}\ A \times B & = & (A, B)
\end{array}
$$

Now, to define a generic version of *encode* it suffices to specify cases for the primitive types, that is, '1', *Char* and *Int*, and for the primitive type constructors, that is, '+' and '×'. The type argument of *encode* is written in angle brackets to distinguish it from the value argument.

$$
\begin{array}{lcl}
encode\langle T\rangle & :: & T \to Bin \\
encode\langle 1\rangle\ () & = & [\,] \\
encode\langle Char\rangle\ c & = & encodeChar\ c \\
encode\langle Int\rangle\ i & = & encodeInt\ i \\
encode\langle A+B\rangle\ (inl\ a) & = & \mathtt{0} : encode\langle A\rangle\ a \\
encode\langle A+B\rangle\ (inr\ b) & = & \mathtt{1} : encode\langle B\rangle\ b \\
encode\langle A\times B\rangle\ (a,b) & = & encode\langle A\rangle\ a +\!\!+\ encode\langle B\rangle\ b
\end{array}
$$

The type signature of *encode*, which is mandatory, makes explicit that the type of $encode\langle T\rangle$ depends on the type argument or *type index* T. Interestingly, each equation is more or less inevitable. To encode the single element of the unit type no bits are required. Integers and characters are encoded using the primitive functions *encodeChar* and *encodeInt*. To encode an element of a sum we emit one bit for the constructor followed by the encoding of its argument. Finally, the encoding of a pair is given by the concatenation of the component's encodings.

This simple definition contains all ingredients needed to compress elements of arbitrary data types. For instance, $encode\langle Sequ\ Int\rangle$ of type $Sequ\ Int \to Bin$ compresses random-access lists with integer elements and $encode\langle List\ (List\ Bool)\rangle$ compresses Boolean matrices. We will see in later chapters that it is possible to specialize $encode\langle T\rangle$ for a given T obtaining essentially the definitions one would have written by hand.

The generic definition of *decodes* follows the same definitional pattern as *encode*.

$$
\begin{array}{lcl}
decodes\langle T\rangle & :: & Bin \to (T, Bin) \\
decodes\langle 1\rangle\ bin & = & ((), bin) \\
decodes\langle Char\rangle\ bin & = & decodesChar\ bin \\
decodes\langle Int\rangle\ bin & = & decodesInt\ bin \\
decodes\langle A+B\rangle\ [\,] & = & error\ \texttt{"decodes"} \\
decodes\langle A+B\rangle\ (\mathtt{0} : bin) & = & \mathbf{let}\ (a, bin') = decodes\langle A\rangle\ bin\ \mathbf{in}\ (inl\ a, bin') \\
decodes\langle A+B\rangle\ (\mathtt{1} : bin) & = & \mathbf{let}\ (b, bin') = decodes\langle B\rangle\ bin\ \mathbf{in}\ (inr\ b, bin') \\
decodes\langle A\times B\rangle\ bin & = & \mathbf{let}\ (a, bin_1) = decodes\langle A\rangle\ bin \\
 & & \quad\ \ (b, bin_2) = decodes\langle B\rangle\ bin_1 \\
 & & \mathbf{in}\ ((a,b), bin_2)
\end{array}
$$

The pair of functions, *encode* and *decodes*, allows to encode and to decode elements of arbitrary user-defined data types.

Generic proofs A generic program enjoys generic properties. Here is a (very desirable) property of the two functions introduced above:

$$decodes\langle T\rangle\ (encode\langle T\rangle\ t) \quad = \quad (t, [\,])$$

for all types T and for all elements t of T. In other words, the implementation is correct: decoding an encoded value yields the original value.

Like the definition of *encode* and *decodes* the proof of this property proceeds by induction on the structure data types. In fact, we have to prove a slightly stronger statement to push the induction through:

$$decodes\langle T\rangle\ (encode\langle T\rangle\ t +\!\!+\ bin) \quad = \quad (t, bin).$$

For simplicity, we assume that we are working in a strict setting (so that the property trivially holds for $t = \perp$).

- **Case** $T = 1$ and $t = ()$:

$$
\begin{aligned}
& \mathit{decodes}\langle 1\rangle\ (\mathit{encode}\langle 1\rangle\ ()\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ \text{definition of } \mathit{encode}\ \} \\
& \mathit{decodes}\langle 1\rangle\ ([\,]\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ \text{definition of } (+\!\!+)\text{: } [\,]\ +\!\!+\ y = y\ \} \\
& \mathit{decodes}\langle 1\rangle\ \mathit{bin} \\
=\ & \quad \{\ \text{definition of } \mathit{decodes}\ \} \\
& ((), \mathit{bin})
\end{aligned}
$$

- **Case** $T = A + B$ and $t = \mathit{inl}\ a$:

$$
\begin{aligned}
& \mathit{decodes}\langle A+B\rangle\ (\mathit{encode}\langle A+B\rangle\ (\mathit{inl}\ a)\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ \text{definition of } \mathit{encode}\ \} \\
& \mathit{decodes}\langle A+B\rangle\ ((\mathtt{0} : \mathit{encode}\langle A\rangle\ a)\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ \text{definition of } (+\!\!+)\text{: } (a : x)\ +\!\!+\ y = a : (x\ +\!\!+\ y)\ \} \\
& \mathit{decodes}\langle A+B\rangle\ (\mathtt{0} : (\mathit{encode}\langle A\rangle\ a\ +\!\!+\ \mathit{bin})) \\
=\ & \quad \{\ \text{definition of } \mathit{decodes}\ \} \\
& \textbf{let}\ (a', \mathit{bin}') = \mathit{decodes}\langle A\rangle\ (\mathit{encode}\langle A\rangle\ a\ +\!\!+\ \mathit{bin})\ \textbf{in}\ (\mathit{inl}\ a', \mathit{bin}') \\
=\ & \quad \{\ \text{ex hypothesi}\ \} \\
& (\mathit{inl}\ a, \mathit{bin}).
\end{aligned}
$$

- **Case** $T = A + B$ and $t = \mathit{inr}\ a$: analogous.
- **Case** $T = A \times B$ and $t = (a, b)$:

$$
\begin{aligned}
& \mathit{decodes}\langle A\times B\rangle\ (\mathit{encode}\langle A\times B\rangle\ (a, b)\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ \text{definition of } \mathit{encode}\ \} \\
& \mathit{decodes}\langle A\times B\rangle\ ((\mathit{encode}\langle A\rangle\ a\ +\!\!+\ \mathit{encode}\langle B\rangle\ b)\ +\!\!+\ \mathit{bin}) \\
=\ & \quad \{\ (+\!\!+) \text{ is associative: } (x\ +\!\!+\ y)\ +\!\!+\ z = x\ +\!\!+\ (y\ +\!\!+\ z)\ \} \\
& \mathit{decodes}\langle A\times B\rangle\ (\mathit{encode}\langle A\rangle\ a\ +\!\!+\ (\mathit{encode}\langle B\rangle\ b\ +\!\!+\ \mathit{bin})) \\
=\ & \quad \{\ \text{definition of } \mathit{decodes}\ \} \\
& \textbf{let}\ (a', \mathit{bin}_1) = \mathit{decodes}\langle A\rangle\ (\mathit{encode}\langle A\rangle\ a\ +\!\!+\ (\mathit{encode}\langle B\rangle\ b\ +\!\!+\ \mathit{bin})) \\
& \qquad (b', \mathit{bin}_2) = \mathit{decodes}\langle B\rangle\ \mathit{bin}_1 \\
& \textbf{in}\ ((a', b'), \mathit{bin}_2) \\
=\ & \quad \{\ \text{ex hypothesi}\ \} \\
& \textbf{let}\ (b', \mathit{bin}_2) = \mathit{decodes}\langle B\rangle\ (\mathit{encode}\langle B\rangle\ b\ +\!\!+\ \mathit{bin}) \\
& \textbf{in}\ ((a, b'), \mathit{bin}_2) \\
=\ & \quad \{\ \text{ex hypothesi}\ \} \\
& ((a, b), \mathit{bin})
\end{aligned}
$$

Generic reasoning complements generic programming in a useful way. The straightforward proof above establishes the correctness of the implementation for all types

T. In fact, conducting a generic proof is often genuinely simpler than conducting a 'monotypic' proof for a particular instance of T. (If you are not convinced, try to prove the above property for $T = \mathit{Sequ}\ \mathit{Char}$ by structural induction.)

1.1.2 Size functions

Many list processing functions can be generalized to arbitrary data types. Consider, for instance, the polymorphic function $\mathit{length} :: \forall A\,.\,\mathit{List}\ A \to \mathit{Int}$, which computes the length of a list. A *length* or rather a *size* function can also be defined for binary random-access lists and, in fact, for every so-called *container type* (Hoogendijk and de Moor 2000). In general, a *size* function of type $\forall A\,.\,T\ A \to \mathit{Int}$ counts the number of values of type A in a given container of type $T\ A$.

Ad-hoc programs Calculating the size of a list is easy:

$$\begin{array}{lcl}
\mathit{sizeList} & :: & \forall A\,.\,\mathit{List}\ A \to \mathit{Int} \\
\mathit{sizeList}\ \mathit{nil} & = & 0 \\
\mathit{sizeList}\ (\mathit{cons}\ a\ \mathit{as}) & = & 1 + \mathit{sizeList}\ \mathit{as}.
\end{array}$$

Interestingly, we will see later that this definition contains all the information necessary to turn *sizeList* into a generic function.

Binary random-access lists are modelled after the binary natural numbers. Therefore, calculating the length of a random-access list corresponds to converting a binary number into an integer.

$$\begin{array}{lcl}
\mathit{sizeSequ} & :: & \forall A\,.\,\mathit{Sequ}\ A \to \mathit{Int} \\
\mathit{sizeSequ}\ \mathit{endS} & = & 0 \\
\mathit{sizeSequ}\ (\mathit{zeroS}\ s) & = & 2 \times \mathit{sizeSequ}\ s \\
\mathit{sizeSequ}\ (\mathit{oneS}\ a\ s) & = & 1 + 2 \times \mathit{sizeSequ}\ s
\end{array}$$

Since the binary representation of n is $\lceil \mathit{lg}\ (n+1) \rceil$ bits long, *sizeSequ* runs in logarithmic time. So it is fast, but unfortunately it fails to be modular. Assume, for the sake of example, that we want to determine the number of characters in an element of type $\mathit{Sequ}\ (\mathit{List}\ \mathit{Char})$. Using *sizeSequ* we can count the number of strings but that does not help. How do we proceed? Now, we could first map *sizeList* on *Sequ* to obtain a sequence of type $\mathit{Sequ}\ \mathit{Int}$, which we then sum up. So for a start we require a mapping function for *Sequ*:

$$\begin{array}{lcl}
\mathit{mapFork} & :: & \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (\mathit{Fork}\ A_1 \to \mathit{Fork}\ A_2) \\
\mathit{mapFork}\ \mathit{mapA}\ (\mathit{fork}\ a_1\ a_2) & = & \mathit{fork}\ (\mathit{mapA}\ a_1)\ (\mathit{mapA}\ a_2) \\
\mathit{mapSequ} & :: & \forall A_1\ A_2\,.\,(A_1 \to A_2) \to (\mathit{Sequ}\ A_1 \to \mathit{Sequ}\ A_2) \\
\mathit{mapSequ}\ \mathit{mapA}\ \mathit{endS} & = & \mathit{endS} \\
\mathit{mapSequ}\ \mathit{mapA}\ (\mathit{zeroS}\ s) & = & \mathit{zeroS}\ (\mathit{mapSequ}\ (\mathit{mapFork}\ \mathit{mapA})\ s) \\
\mathit{mapSequ}\ \mathit{mapA}\ (\mathit{oneS}\ a\ s) & = & \mathit{oneS}\ (\mathit{mapA}\ a)\ (\mathit{mapSequ}\ (\mathit{mapFork}\ \mathit{mapA})\ s).
\end{array}$$

Note that both *mapFork* and *mapSequ* follow closely the structure of the corresponding type definitions. In fact, we will see later that mapping functions also enjoy a generic definition (Section 3.2.1).

Summing up a sequence of integers is quite tricky:

$$
\begin{array}{lcl}
\mathit{sumFork} & :: & \mathit{Fork}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sumFork}\ (\mathit{fork}\ a_1\ a_2) & = & a_1 + a_2 \\
\mathit{sumSequ} & :: & \mathit{Sequ}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sumSequ}\ \mathit{endS} & = & 0 \\
\mathit{sumSequ}\ (\mathit{zeroS}\ s) & = & \mathit{sumSequ}\ (\mathit{mapSequ}\ \mathit{sumFork}\ s) \\
\mathit{sumSequ}\ (\mathit{oneS}\ a\ s) & = & a + \mathit{sumSequ}\ (\mathit{mapSequ}\ \mathit{sumFork}\ s).
\end{array}
$$

Consider the last equation of *sumSequ* where we have to sum up the sequence s of type *Sequ* (*Fork Int*). We proceed roughly as before: first we map *sumFork* on *Sequ* to obtain an element of type *Sequ Int* which we then recursively sum up. We can now solve the original problem: $\mathit{sumSequ} \cdot \mathit{mapSequ}\ \mathit{sizeList}$ counts the number of characters in an element of type *Sequ* (*List Char*).

The above solution is quite involved. Let us pursue an alternative approach. The recursion scheme of *encodeSequ* and *decodeSequ* suggests to parameterize *sizeSequ* by a function that calculates the 'size' of an element.

$$
\begin{array}{lcl}
\mathit{countFork} & :: & \forall A\,.\,(A \to \mathit{Int}) \to (\mathit{Fork}\ A \to \mathit{Int}) \\
\mathit{countFork}\ \mathit{countA}\ (\mathit{fork}\ a_1\ a_2) & = & \mathit{countA}\ a_1 + \mathit{countA}\ a_2 \\
\mathit{countSequ} & :: & \forall A\,.\,(A \to \mathit{Int}) \to (\mathit{Sequ}\ A \to \mathit{Int}) \\
\mathit{countSequ}\ \mathit{countA}\ \mathit{endS} & = & 0 \\
\mathit{countSequ}\ \mathit{countA}\ (\mathit{zeroS}\ s) & = & \mathit{countSequ}\ (\mathit{countFork}\ \mathit{countA})\ s \\
\mathit{countSequ}\ \mathit{countA}\ (\mathit{oneS}\ a\ s) & = & \mathit{countA}\ a + \mathit{countSequ}\ (\mathit{countFork}\ \mathit{countA})\ s
\end{array}
$$

This style probably looks familiar by now. Consider again the last equation: to sum up the sequence s of type *Sequ* (*Fork A*) we call *countSequ* (*countFork countA*). Note that *sizeSequ* and *countSequ* are related by $\mathit{countSequ}\ \mathit{countA} = \mathit{sumSequ} \cdot \mathit{mapSequ}\ \mathit{countA}$.

The parameterized version of *sizeSequ* is quite versatile. If we pass the constant function $k\ 1$ to *countSequ* we obtain (a linear-time variant of) *sizeSequ*. Passing the identity function yields *sumSequ*:

$$
\begin{array}{lcl}
\mathit{sizeSequ}' & :: & \forall A\,.\,\mathit{Sequ}\ A \to \mathit{Int} \\
\mathit{sizeSequ}' & = & \mathit{countSequ}\ (k\ 1) \\
\mathit{sumSequ}' & :: & \mathit{Sequ}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sumSequ}' & = & \mathit{countSequ}\ \mathit{id} \\
\mathit{sizeSequList} & :: & \forall A\,.\,\mathit{Sequ}\ (\mathit{List}\ A) \to \mathit{Int} \\
\mathit{sizeSequList} & = & \mathit{countSequ}\ \mathit{sizeList}.
\end{array}
$$

It is interesting if not revealing to compare *sumSequ* and $\mathit{sumSequ}'$. Recall that an element of type *Sequ* is a sequence of perfect binary leaf trees. The first function processes the trees bottom-up: in each recursive step the nodes on the lowest level are summed up (using *mapSequ sumFork*). By contrast, $\mathit{sumSequ}'$ operates in two stages: while recursing *countSequ* constructs a tailor-made function of type $\mathit{Fork}^i\ A \to \mathit{Int}$, which when applied reduces a perfect binary leaf tree in a single top-down pass. Clearly, the latter algorithm is more efficient than the former.

A generic program The semantics of the *size* function for a container type T is crystal clear: it counts the number of elements of type A in a given value of type $T\ A$. This suggests that we should be able to program a generic function $\mathit{size}\langle T\rangle :: \forall A\,.\,T\ A \to \mathit{Int}$, which works for all T. Note that the type signature of *size* is more involved than the signature of *encode* since *size* is indexed by a type constructor rather than by a type. The type of *size* ensures that we can determine the size of a list or a binary random-access list but not the size of a character or an integer. Now, in order to define $size\langle T\rangle$ generically for all T we must explicate the structure of type constructors such as *List*, *Fork* and *Sequ*.

It turns out that we have to consider only one additional case, the identity type given by $\Lambda X \,.\, X$. Here the upper-case lambda denotes abstraction on the type level. Consequently, the generic *size* function is uniquely determined by the following equations.

$$
\begin{array}{lcl}
size\langle T\rangle & :: & \forall A \,.\, T\ A \to Int \\
size\langle \Lambda X \,.\, X\rangle\ a & = & 1 \\
size\langle \Lambda X \,.\, 1\rangle\ u & = & 0 \\
size\langle \Lambda X \,.\, Char\rangle\ c & = & 0 \\
size\langle \Lambda X \,.\, Int\rangle\ i & = & 0 \\
size\langle \Lambda X \,.\, F\ X + G\ X\rangle\ (inl\ f) & = & size\langle F\rangle\ f \\
size\langle \Lambda X \,.\, F\ X + G\ X\rangle\ (inr\ g) & = & size\langle G\rangle\ g \\
size\langle \Lambda X \,.\, F\ X \times G\ X\rangle\ (f, g) & = & size\langle F\rangle\ f + size\langle G\rangle\ g
\end{array}
$$

The type patterns on the left-hand side involve type abstractions since *size* is parameterized by a type constructor. Consider, for example, the type pattern $\Lambda X \,.\, F\ X \times G\ X$. The type variables F and G range over type constructors. The corresponding instance $size\langle \Lambda X \,.\, F\ X \times G\ X\rangle :: \forall A \,.\, F\ A \times G\ A \to Int$ can then be inductively defined in terms of $size\langle F\rangle :: \forall A \,.\, F\ A \to Int$ and $size\langle G\rangle :: \forall A \,.\, G\ A \to Int$. Let us consider each equation in turn. A container of type $(\Lambda X \,.\, X)\ A = A$ includes exactly only one element of type A; a container of type $(\Lambda X \,.\, 1)\ A = 1$, $(\Lambda X \,.\, Char)\ A = Char$ or $(\Lambda X \,.\, Int)\ A = Int$ includes no elements of type A. To determine the size of a container of type $(\Lambda X \,.\, F\ X + G\ X)\ A = F\ A + G\ A$ we must either calculate the size of a container of type $F\ A$ or that of a container of type $G\ A$ depending on which component of the sum the argument comes from. Finally, the size of a container of type $(\Lambda X \,.\, F\ X \times G\ X)\ A = F\ A \times G\ A$ is given by the sum of the size of the two components.

Note that the *sizeList* instance provides all the necessary information for defining the generic *size* function, since the definition of the list data type, $List = \Lambda X \,.\, 1 + X \times List\ X$, involves the identity type, the unit type, a sum and a product.

The generic definition allows us to determine the size of containers of arbitrary types. For instance, $size\langle \Lambda A \,.\, Sequ\ (List\ A)\rangle$ calculates the number of elements in a sequence of lists. If we specialize this instance we obtain a definition similar to *sizeSequList*. Unfortunately, specializing $size\langle Sequ\rangle$ yields the linear-time *sizeSequ'* and not the logarithmic *sizeSequ*. In general, a 'structure-strict', generic function has at least a linear running time. So we cannot reasonably expect to achieve the efficiency of a handcrafted implementation that exploits data-structural invariants. However, we will see later that we can derive *sizeSequ* from the generic definition in a systematic way (Section 4.1.2).

Ad-hoc versus generic programs Giving ad-hoc definitions of functions like *encode*, *decodes* and *size* is sometimes simple and sometimes involving. While the generic definition is slightly more abstract, it is also to a high degree inevitable. It is this feature that makes generic programming light and sweet. Further still, the generic programmer need not deal with type abstraction and type recursion. Genericity provides these cases 'for free'.

1.2 Overview

This thesis shows how to program and reason generically. We look at several examples of generic programs and proofs and describe an extension to Haskell that supports generic programming.

Chapter 2 provides the necessary background for reading this thesis. We sketch Haskell's type and class system and introduce the simply typed and the polymorphic λ-calculus. The polymorphic λ-calculus is used as the formal basis for the generic programming extension.

Chapter 3 shows how to define generic values and explains how to specialize a generic value to concrete instances of data types. We consider in increasing level of difficulty: values such as *encode* that are indexed by types, values such as *size* that are indexed by type constructors and finally values that are indexed by type constructors of arbitrary kinds. The specialization is such that neither run-time passing of types nor case analysis on types is required. This chapter is based on the papers "A new approach to generic functional programming" (Hinze 2000e) and "Polytypic values possess polykinded types" (Hinze 2000g).

Chapter 4 which is concerned with generic reasoning introduces two generic proof methods. The first method is a variant of fixed point induction. It can also be used constructively to derive a generic program from its specification. The second method builds on so-called logical relations. This chapter is based on the papers "Polytypic programming with ease" (Hinze 2000f) and "Polytypic values possess polykinded types" (Hinze 2000g).

Chapter 5 presents several examples of generic functions and associated generic properties. In particular, we discuss generic implementations of dictionaries and memo tables based on generalized tries. Generalized tries make a particularly interesting application of generic programming since they can be modelled as a type-indexed data type. This chapter is based on the papers "Generalizing generalized tries" (Hinze 2000b) and "Memo functions, polytypically!" (Hinze 2000d).

Chapter 6 describes an extension to Haskell that supports generic programming. We discuss the implementation and identify several extensions that are useful in a practical setting. This chapter is based on the papers "A generic programming extension for Haskell" (Hinze 1999) and "Derivable type classes" (Hinze and Peyton Jones 2000).

Chapter 2

Background

This chapter reviews background material that is needed in subsequent chapters.

All of the example programs in this thesis will be given in the functional programming language Haskell 98 (Peyton Jones and Hughes 1999). I generally assume a passing familiarity with Haskell, its syntax and semantics. There are several excellent textbooks on Haskell, which the reader may wish to consult. I heartily recommend (Bird 1998), (Thompson 1999) and (Hudak 2000). However, since types play a central rôle in this thesis, I will discuss Haskell's type system (Section 2.1) and its class system (Section 2.2) in some detail.

In the introduction we have already seen that generic programs are defined by giving cases for the primitive type constructors '1', '+', '×' etc. Section 2.3 provides a more abstract view of these type constructors and introduces a set of combinators that we will often use to define generic programs in a point-free style.

While we employ Haskell for the practical part, the language of choice for the theoretical part is the polymorphic λ-calculus. To prepare the ground we first introduce the simply typed λ-calculus in Section 2.4 and then the polymorphic λ-calculus in Section 2.5.

2.1 The type system of Haskell

Haskell offers one basic construct for defining new types: a so-called *data type declaration*. In general, a **data** declaration has the following form:

$$\mathbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}.$$

This definition simultaneously introduces a new type constructor B and n value constructors $k_1, \ldots, k_n$, whose types are given by

$$k_j \quad :: \quad \forall A_1\ \ldots\ A_m\,.\,T_{j1} \to \cdots \to T_{jm_j} \to B\ A_1\ \ldots\ A_m.$$

The type parameters $A_1, \ldots, A_m$ must be distinct and may appear on the right-hand side of the declaration. If $m > 0$, then B is called a *parameterized type*. Data type declarations can be recursive, that is, D may also appear on the right-hand side. In general, data types are defined by a system of mutually recursive data type declarations.

The following sections provide numerous examples of data type declarations organized in increasing order of difficulty.

REMARK 2.1 Haskell also offers type synonym declarations of the form

$$\mathbf{type}\ B\ A_1\ \ldots\ A_m \quad = \quad T$$

and data type renamings of the form

$$\mathbf{newtype}\ B\ A_1\ \ldots\ A_m \quad = \quad k\ T.$$

A type synonym introduces a type that is equivalent to the type on the right-hand side, that is, $B\ A_1\ \ldots\ A_m$ merely serves as an abbreviation for T. A data type renaming introduces a new distinct type whose representation is the same as the type on the right-hand side. The constructor k is used to coerce between the new and the old type. □

2.1.1 Finite types

Data type declarations subsume enumerated types. In this special case, we only have nullary value constructors, that is, $m_1 = \cdots = m_n = 0$. The following declaration defines a simple enumerated type, the type of truth values.

$$\textbf{data}\ \mathit{Bool} \quad = \quad \mathit{false} \mid \mathit{true}$$

Data type declarations also subsume record types. In this case, we have only one value constructor, that is, $n = 1$.

$$\textbf{data}\ \mathit{Fork}\ A \quad = \quad \mathit{fork}\ A\ A$$

An element of *Fork* A is a pair whose two components both have type A. Haskell assigns a *kind* to each type constructor. One can think of a kind as the 'type' of a type constructor. The type constructor *Fork* defined above has kind $\star \to \star$. The '$\star$' kind represents nullary constructors like *Char*, *Int* or *Bool*. The kind $\mathfrak{T} \to \mathfrak{U}$ represents type constructors that map type constructors of kind $\mathfrak{T}$ to those of kind $\mathfrak{U}$. Note that the term 'type' is sometimes used for nullary type constructors.

The following types can be used to represent 'optional values'.

$$\begin{aligned}
\textbf{data}\ \mathit{Maybe}\ A &\quad = \quad \mathit{nothing} \mid \mathit{just}\ A \\
\textbf{data}\ A \times_{\bullet} B &\quad = \quad \mathit{null} \mid \mathit{pair}\ A\ B
\end{aligned}$$

An element of type *Maybe* A is an 'optional A': it is either of the form *nothing* or of the form *just* a where a is of type A. Elements of type $A \times_{\bullet} B$ are called optional pairs. The type constructor *Maybe* has kind $\star \to \star$ and $(\times_{\bullet})$ has kind $\star \to (\star \to \star)$. Perhaps surprisingly, binary type constructors like $(\times_{\bullet})$ are, in fact, curried in Haskell.

2.1.2 Regular types

A simple recursive data type is the type of natural numbers.

$$\textbf{data}\ \mathit{Nat} \quad = \quad \mathit{zero} \mid \mathit{succ}\ \mathit{Nat}$$

The number 6, for instance, is given by

$$\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))))).$$

The following alternative definition of the natural numbers is based on the binary number system.

$$\textbf{data}\ \mathit{BNat} \quad = \quad \mathit{endB} \mid \mathit{zeroB}\ \mathit{BNat} \mid \mathit{oneB}\ \mathit{BNat}$$

Using this representation the number $6 = (011)_2$ reads (the bits are written from least significant to most significant):

$$\mathit{zeroB}\ (\mathit{oneB}\ (\mathit{oneB}\ \mathit{endB})).$$

The most popular data type is without doubt the type of parametric lists:

$$\mathbf{data}\ \mathit{List}\ A \quad = \quad \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A).$$

The empty list is denoted *nil*; $\mathit{cons}\ a\ x$ denotes the list whose first element is a and whose remaining elements are those of x. The list of the first six prime numbers, for instance, is given by

$$\mathit{cons}\ 2\ (\mathit{cons}\ 3\ (\mathit{cons}\ 5\ (\mathit{cons}\ 7\ (\mathit{cons}\ 11\ (\mathit{cons}\ 13\ \mathit{nil}))))).$$

Haskell provides special syntax for lists: $\mathit{List}\ A$ is written $[A]$ (the type constructor *List* in isolation is written '$[\,]$'), *nil* and $\mathit{cons}\ a\ x$ are written $[\,]$ and $a : x$, respectively. We already made use of this notation in the introduction. The operator for list concatenation, also employed in the introduction, is defined

$$\begin{array}{lcl}
(+\!\!+) & :: & \forall A\,.\,[A] \to [A] \to [A] \\
[\,] +\!\!+\ y & = & y \\
(a : x) +\!\!+\ y & = & a : (x +\!\!+\ y).
\end{array}$$

The function $(+\!\!+)$ has a *polymorphic type*: $[A] \to [A] \to [A]$ is a legal type for all instances of the type variable A. Recall that we write polymorphic types using explicit universal quantifiers, though this is not legal Haskell 98 syntax. In Haskell 98 type variables are implicitly quantified: the type signature of list concatenation is just $(+\!\!+) :: [A] \to [A] \to [A]$.

In the sequel we require the function *wrap* that turns an element into a singleton list:

$$\begin{array}{lcl}
\mathit{wrap} & :: & \forall A\,.\,A \to [A] \\
\mathit{wrap}\ a & = & [a].
\end{array}$$

The following definition introduces binary external search trees.

$$\mathbf{data}\ \mathit{Tree}\ A\ B \quad = \quad \mathit{leaf}\ A \mid \mathit{node}\ (\mathit{Tree}\ A\ B)\ B\ (\mathit{Tree}\ A\ B)$$

We distinguish between external nodes of the form $\mathit{leaf}\ a$ and internal nodes of the form $\mathit{node}\ l\ b\ r$. The former are labelled with elements of type A while the latter are labelled with elements of type B. Here is an example element of type $\mathit{Tree}\ \mathit{Bool}\ \mathit{Int}$:

$$\mathit{node}\ (\mathit{leaf}\ \mathit{true})\ 7\ (\mathit{node}\ (\mathit{leaf}\ \mathit{true})\ 9\ (\mathit{leaf}\ \mathit{false})).$$

The following data type declaration captures multiway branching trees, also known as *rose trees* (Bird 1998).

$$\mathbf{data}\ \mathit{Rose}\ A \quad = \quad \mathit{branch}\ A\ (\mathit{List}\ (\mathit{Rose}\ A))$$

A node is labelled with an element of type A and has a list of subtrees. An example element of type $\mathit{Rose}\ \mathit{Int}$ is:

$$\begin{array}{l}
\mathit{branch}\ 2\ (\mathit{cons}\ (\mathit{branch}\ 3\ \mathit{nil}) \\
\qquad\qquad (\mathit{cons}\ (\mathit{branch}\ 5\ \mathit{nil}) \\
\qquad\qquad (\mathit{cons}\ (\mathit{branch}\ 7\ (\mathit{cons}\ (\mathit{branch}\ 11\ \mathit{nil}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{cons}\ (\mathit{branch}\ 13\ \mathit{nil})\ \mathit{nil})))\ \mathit{nil}))).
\end{array}$$

The type *Rose* falls back on the type *List*. Instead, we may introduce *Rose* using two mutually recursive data type declarations:

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{Rose}'\ A & = & \mathit{branch}'\ A\ (\mathit{Forest}\ A) \\
\mathbf{data}\ \mathit{Forest}\ A & = & \mathit{nilF} \mid \mathit{consF}\ (\mathit{Rose}'\ A)\ (\mathit{Forest}\ A).
\end{array}$$

Now $Rose'$ depends on $Forest$ and vice versa.

The type parameters of a data type may range over type constructors of arbitrary kinds.[1] The following generalization of rose trees, that abstracts over the $List$ data type, illustrates this feature.

$$\textbf{data}\ GRose\ F\ A \quad = \quad gbranch\ A\ (F\ (GRose\ F\ A))$$

A slight variant of this definition has been used by Okasaki (1998) to extend an implementation of priority queues with an efficient merge operation. The type constructor $GRose$ has kind $(\star \to \star) \to (\star \to \star)$, that is, $GRose$ has a so-called *second-order kind* where the order of a kind is given by

$$\begin{aligned} order(\star) &= 0 \\ order(\mathfrak{T} \to \mathfrak{U}) &= max\{1 + order(\mathfrak{T}), order(\mathfrak{U})\}. \end{aligned}$$

Applying $GRose$ to $List$ yields the type of rose trees.

The following data type declaration introduces a fixed point operator on the level of types. This definition appears, for instance, in (Meijer and Hutton 1995) where it is employed to give a generic definition of so-called *cata-* and *anamorphisms*.

$$\begin{aligned} \textbf{newtype}\ Fix\ F &= in\ (F\ (Fix\ F)) \\ \textbf{data}\ ListBase\ A\ B &= nilL \mid consL\ A\ B \end{aligned}$$

The kinds of these type constructors are $Fix :: (\star \to \star) \to \star$ and $ListBase :: \star \to (\star \to \star)$. Using Fix and $ListBase$ the data type of parametric lists can alternatively be defined by

$$\textbf{type}\ List\ A \quad = \quad Fix\ (ListBase\ A).$$

Here is the list of the first six prime numbers written as an element of type $Fix\ (ListBase\ Int)$:

$$\begin{aligned} &in\ (consL\ 2\ (in\ (consL\ 3\ (in\ (consL\ 5 \\ &\qquad (in\ (consL\ 7\ (in\ (consL\ 11\ (in\ (consL\ 13\ (in\ nilL)))))))\ ))))). \end{aligned}$$

2.1.3 Nested types

A *regular* or *uniform* data type is a parameterized type whose definition does not involve a change of the type parameter(s). The data types of the previous section are without exception regular types. This section is concerned with non-regular or *nested* types (Bird and Meertens 1998). We have already remarked that nested data types are practically important since they can capture data-structural invariants in a way that regular data types cannot. The following data type declaration, for instance, defines perfectly balanced, binary leaf trees (Hinze 2000a)—perfect trees for short.

$$\textbf{data}\ Perfect\ A \quad = \quad zeroP\ A \mid succP\ (Perfect\ (Fork\ A))$$

This equation can be seen as a bottom-up definition of perfect trees: a perfect tree is either a singleton tree or a perfect tree that contains pairs of elements. Here is a perfect tree of type $Perfect\ Int$:

$$\begin{aligned} succP\ (succP\ (succP\ (zeroP\ (fork\ (&fork\ (fork\ 2\ 3) \\ &\qquad (fork\ 5\ 7)) \\ &(fork\ (fork\ 11\ 13) \\ &\qquad (fork\ 17\ 19))\ )))). \end{aligned}$$

[1] Note that Miranda (trademark of Research Software Ltd), Standard ML, and previous versions of Haskell (1.2 and before) only have first-order kinded data types.

Note that the height of the perfect tree is encoded in the prefix of *succP* and *zeroP* constructors.

In the introduction we have already encountered Okasaki's type of binary random-access lists (1998).

$$\begin{array}{lcl} \textbf{data}\ \mathit{Sequ}\ A & = & \mathit{endS} \\ & | & \mathit{zeroS}\ (\mathit{Sequ}\ (\mathit{Fork}\ A)) \\ & | & \mathit{oneS}\ A\ (\mathit{Sequ}\ (\mathit{Fork}\ A)) \end{array}$$

Recall that this definition captures the invariant that binary random-access lists are sequences of perfect trees stored in increasing order of height. Using this representation the sequence of the first six prime numbers reads:

$$\mathit{zeroS}\ (\mathit{oneS}\ (\mathit{fork}\ 2\ 3)\ (\mathit{oneS}\ (\mathit{fork}\ (\mathit{fork}\ 5\ 7)\ (\mathit{fork}\ 11\ 13))\ \mathit{endS})).$$

REMARK 2.2 Binary random-access lists are modelled after the binary number system (while ordinary lists are modelled after the unary representation of the natural numbers). For instance, 'consing' an element to a random-access list corresponds to incrementing a binary number:

$$\begin{array}{lcl} \mathit{incB} & :: & \mathit{BNat} \to \mathit{BNat} \\ \mathit{incB}\ \mathit{endB} & = & \mathit{oneB}\ \mathit{endB} \\ \mathit{incB}\ (\mathit{zeroB}\ b) & = & \mathit{oneB}\ b \\ \mathit{incB}\ (\mathit{oneB}\ b) & = & \mathit{zeroB}\ (\mathit{incB}\ b) \\ \mathit{consS} & :: & \forall A\,.\,A \to \mathit{Sequ}\ A \to \mathit{Sequ}\ A \\ \mathit{consS}\ a\ \mathit{endS} & = & \mathit{oneS}\ a\ \mathit{endS} \\ \mathit{consS}\ a\ (\mathit{zeroS}\ s) & = & \mathit{oneS}\ a\ s \\ \mathit{consS}\ a\ (\mathit{oneS}\ a'\ s) & = & \mathit{zeroS}\ (\mathit{consS}\ (\mathit{fork}\ a\ a')\ s). \end{array}$$

For a more in-depth treatment of the correspondence between number systems and container types the reader is referred to (Okasaki 1998; Hinze 2000c). □

The types *Perfect* and *Sequ* are examples of so-called *linear nests*: the parameters of the recursive calls do not themselves contain occurrences of the defined type. A non-linear nest is the following type taken from (Bird and Meertens 1998):

$$\textbf{data}\ \mathit{Bush}\ A \quad = \quad \mathit{nilB} \mid \mathit{consB}\ A\ (\mathit{Bush}\ (\mathit{Bush}\ A)).$$

An element of type *Bush A* resembles an ordinary list except that the i-th element hast type $\mathit{Bush}^i\ A$ rather than A. Here is an example element of type *Bush Int*:

$$\begin{array}{l} \mathit{consB}\ 1\ (\mathit{consB}\ (\mathit{consB}\ 2\ \mathit{nilB}) \\ \qquad\qquad (\mathit{consB}\ (\mathit{consB}\ (\mathit{consB}\ 3\ \mathit{nilB})\ \mathit{nilB})\ \mathit{nilB})). \end{array}$$

Perhaps surprisingly, we will get to know a practical application of this data type in Section 5.5, which deals with so-called *generalized tries*.

Finally, let us take a look at some *higher-order nests* where the type parameter that is instantiated in a recursive call ranges over type constructors rather than types.

$$\begin{array}{lcl} \textbf{data}\ \mathit{FMapFork}\ \mathit{FA}\ V & = & \mathit{trieFork}\ (\mathit{FA}\ (\mathit{FA}\ V)) \\ \textbf{data}\ \mathit{FMapSequ}\ \mathit{FA}\ V & = & \mathit{nullSequ} \\ & | & \mathit{trieSequ}\ (\mathit{Maybe}\ V) \\ & & \qquad\qquad (\mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})\ V) \\ & & \qquad\qquad (\mathit{FA}\ (\mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})\ V)) \end{array}$$

The types $FMapFork, FMapSequ :: (\star \to \star) \to (\star \to \star)$ represent the generalized tries for *Fork* and *Sequ*. These types will be explained in Section 5.5. Note that the type constructor *FMapFork* is the type-level counterpart of the function $twice\ f\ x = f\ (f\ x)$, which applies a given function twice to a given value.

Here is another example of a nested data type of second-order kind:

$$
\begin{array}{lcl}
\textbf{type}\ Square\ A & = & Square'\ Nil\ A \\
\textbf{data}\ Square'\ F\ A & = & zeroM\ (F\ (F\ A)) \mid succM\ (Square'\ (Cons\ F)\ A) \\
\textbf{data}\ Nil\ A & = & nilN \\
\textbf{data}\ Cons\ F\ A & = & consC\ A\ (F\ A).
\end{array}
$$

The type constructors have kinds $Square, Nil :: \star \to \star$ and $Square', Cons :: (\star \to \star) \to (\star \to \star)$. The type *Square* implements square $n \times n$ matrices (Okasaki 1999; Hinze 2000c). In contrast to common representations, such as lists of lists, the 'squareness' constraint is automatically enforced by the type system. As an example, here is a square matrix of size 3:

$$
\begin{array}{l}
succM\ (succM\ (succM\ (zeroM \\
\qquad (consC\ (consC\ a_{11}\ (consC\ a_{12}\ (consC\ a_{13}\ nilN))) \\
\qquad (consC\ (consC\ a_{21}\ (consC\ a_{22}\ (consC\ a_{23}\ nilN))) \\
\qquad (consC\ (consC\ a_{31}\ (consC\ a_{32}\ (consC\ a_{33}\ nilN))) \\
\qquad (nilN))))\ ))).
\end{array}
$$

Note that the dimension of the matrix is encoded in the prefix of *succM* and *zeroM* constructors.

2.1.4 Functional types

Data types may also contain functional types as the following declaration taken from (Hallgren and Carlsson 1995) illustrates.

$$
\textbf{data}\ SP\ A\ B \quad = \quad put\ B\ (SP\ A\ B) \mid get\ (A \to SP\ A\ B)
$$

The name *SP* stands for 'stream processor'. Think of an element of type $SP\ A\ B$ as a process that receives messages of type A and sends messages of type B. Here is a simple stream processor that resends each ingoing message twice.

$$
\begin{array}{lcl}
double & :: & \forall A\,.\, SP\ A\ A \\
double & = & get\ (\lambda a \to put\ a\ (put\ a\ double))
\end{array}
$$

As another example, consider the operator $(\ggg)$ that serially composes two stream processors.

$$
\begin{array}{lcl}
(\ggg) & :: & \forall A\ B\ C\,.\, SP\ A\ B \to SP\ B\ C \to SP\ A\ C \\
sp_1 \ggg put\ c\ sp_2 & = & put\ c\ (sp_1 \ggg sp_2) \\
put\ b\ sp_1 \ggg get\ fsp_2 & = & sp_1 \ggg fsp_2\ b \\
get\ fsp_1 \ggg sp_2 & = & get\ (\lambda a \to fsp_1\ a \ggg sp_2)
\end{array}
$$

For instance, $double \ggg double$ is a stream processor that resends each ingoing message four times.

2.2 The class system of Haskell

2.2.1 Type classes

The major innovation of Haskell is its support for overloading, based on type classes. For example, the Haskell Prelude defines the class *Eq*:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ A\ \textbf{where} \\
\quad \begin{array}{lcl}
(==), (\neq) & :: & A \to A \to \mathit{Bool} \\
a_1 \neq a_2 & = & \mathit{not}\ (a_1 == a_2) \\
a_1 == a_2 & = & \mathit{not}\ (a_1 \neq a_2).
\end{array}
\end{array}
$$

This *class declaration* defines two overloaded top-level functions, called *methods*, whose types are

$$(==), (\neq) \quad :: \quad \forall A \,.\, (\mathit{Eq}\ A) \Rightarrow A \to A \to \mathit{Bool}.$$

Before we can use $(==)$ on values of, say *Int*, we must explain how to take equality over *Int* values:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Int}\ \textbf{where} \\
\quad (==) \quad = \quad \mathit{equalInt}.
\end{array}
$$

Here we suppose that $\mathit{equalInt} :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$ is provided from somewhere. The *instance declaration* makes *Int* an element of the type class *Eq* and says 'the $(==)$ function at type *Int* is implemented by *equalInt*'. The $(\neq)$ method need not be explicitly defined since the class definition provides a *default declaration* for $(\neq)$: it is simply the negation of the code for $(==)$. In fact, the class declaration specifies default methods for both $(==)$ and $(\neq)$. So you can *either* give a definition for $(==)$, *or* a definition for $(\neq)$, or both. However, if you specify neither, then you will get an infinite loop.

How can we take equality of lists of values? Two lists are equal if they have the same length and corresponding elements are equal. Hence, we require equality over the element type:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Eq}\ A) \Rightarrow \mathit{Eq}\ (\mathit{List}\ A)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{nil} == \mathit{nil} & = & \mathit{true} \\
\mathit{nil} == \mathit{cons}\ a_2\ x_2 & = & \mathit{false} \\
\mathit{cons}\ a_1\ x_1 == \mathit{nil} & = & \mathit{false} \\
\mathit{cons}\ a_1\ x_1 == \mathit{cons}\ a_2\ x_2 & = & a_1 == a_2 \wedge x_1 == x_2.
\end{array}
\end{array}
$$

This instance declaration says 'if A is an instance of *Eq*, then *List* A is an instance of *Eq*, as well'.

Though type classes bear a strong resemblance to generic definitions, they do not support generic programming. A type class declaration corresponds roughly to the type signature of a generic definition—or rather, to a collection of type signatures. Instance declarations are related to the type cases of a generic definition. The crucial difference is that a generic definition works for all types, whereas instance declarations must be explicitly provided by the programmer for each newly defined data type. There is, however, one exception to this rule. For a handful of built-in classes Haskell provides special support, the so-called '**deriving**' mechanism. For instance, if you define

$$\textbf{data}\ \mathit{List}\ A \quad = \quad \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A)\ \textbf{deriving}\ (\mathit{Eq})$$

then Haskell generates the 'obvious' code for equality. What 'obvious' means is specified informally in an Appendix of the language definition (Peyton Jones and Hughes 1999).

REMARK 2.3 The idea suggests itself to use generic definitions for specifying default methods so that the programmer can define her own derivable classes. This idea is pursued further in Hinze and Peyton Jones (2000). □

2.2.2 Constructor classes

Type classes may also abstract over type constructors, in which case they are called *constructor classes* (Jones 1995). For instance, the Haskell Prelude defines the class *Functor*:

$$\begin{aligned} &\mathbf{class}\ \mathit{Functor}\ F\ \mathbf{where} \\ &\quad \mathit{fmap} \ :: \ \forall A\ B\,.\,(A \to B) \to (F\ A \to F\ B). \end{aligned}$$

The method *fmap* is the so-called *mapping function* for the data type F. The mapping function applies a given function to each element of type A in a given container of type $F\ A$. We have already encountered the mapping functions of the data types *List*, *Fork* and *Sequ* in the introduction. Here is the mapping function of *List* rephrased as an instance of *Functor*:

$$\begin{aligned} &\mathbf{instance}\ \mathit{Functor}\ \mathit{List}\ \mathbf{where} \\ &\quad \mathit{fmap}\ f\ \mathit{nil} &&= \mathit{nil} \\ &\quad \mathit{fmap}\ f\ (\mathit{cons}\ a\ \mathit{as}) &&= \mathit{cons}\ (f\ a)\ (\mathit{fmap}\ f\ \mathit{as}). \end{aligned}$$

The term 'functor' stems from a branch of mathematics called *category theory*, which is concerned with the study of algebraic structure. I will say more about category theory in Section 2.3. For the moment let me only remark that every instance of *Functor* should satisfy the so-called *functor laws*:

$$\begin{aligned} \mathit{fmap}\ \mathit{id} &= \mathit{id} && \text{(functor law)} \\ \mathit{fmap}\ (f \cdot g) &= \mathit{fmap}\ f \cdot \mathit{fmap}\ g. && \text{(—— " ——)} \end{aligned}$$

That is, *fmap* respects identity and composition.

Another important example of a constructor class is the *Monad* class. Again, monads have their roots in category theory. In the early nineties Moggi proposed them as a means to structure denotational semantics (1990, 1991). Wadler popularized Moggi's idea in the functional programming community by using monads to structure functional programs (1990, 1992, 1995). In Haskell monads are captured by the following class definition.

$$\begin{aligned} &\mathbf{class}\ \mathit{Monad}\ M\ \mathbf{where} \\ &\quad \mathit{return} &&:: \forall A\,.\,A \to M\ A \\ &\quad (\ggg\!=) &&:: \forall A\ B\,.\,M\ A \to (A \to M\ B) \to M\ B \\ &\quad (\gg) &&:: \forall A\ B\,.\,M\ A \to M\ B \to M\ B \\ &\quad \mathit{fail} &&:: \forall A\,.\,\mathit{String} \to M\ A \\ &\quad m \gg n &&= m \ggg\!= k\ n \\ &\quad \mathit{fail}\ s &&= \mathit{error}\ s \end{aligned}$$

The essential idea of monads is to distinguish between *computations* and *values*. This distinction is reflected on the type level: an element of $M\ A$ represents a computation that yields a value of type A. A computation may involve, for instance, state, exceptions, or nondeterminism.

The trivial computation that immediately returns the value a is denoted by $\mathit{return}\ a$. The operator $(\gg=)$, commonly called 'bind', combines two computations: $m \gg= k$ applies k to the result of the computation m. The derived operation $(\gg)$ provides a handy shortcut if one is not interested in the result of the first computation. The operation *fail* is used for signaling error conditions. Note that *fail* does not stem from the mathematical concept of a monad, but has been added to the monad class for pragmatic reasons, see (Peyton Jones and Hughes 1999, Section 3.14).

The operations are required to satisfy the following so-called *monad laws*.

$$
\begin{array}{lcll}
\mathit{return}\ a \gg= k & = & k\ a & \text{(monad law)} \\
m \gg= \mathit{return} & = & m & \text{(—— " ——)} \\
(m \gg= k_1) \gg= k_2 & = & m \gg= (\lambda a \to k_1\ a \gg= k_2) & \text{(—— " ——)}
\end{array}
$$

Roughly speaking, *return* is the unit of $(\gg=)$ and $(\gg=)$ is associative. The monoidal structure becomes more apparent if the laws are rephrased in terms of the monadic composition, see below.

Several data types have a computational content. For instance, the type *Maybe* can be used to model exceptions: *just a* represents a 'normal' or successful computation yielding the value a while *nothing* represents an exceptional or failing computation. The following instance declaration shows how to define *return* and $(\gg=)$ for *Maybe*.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{Maybe}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{return} & = & \mathit{just} \\
\mathit{nothing} \gg= k & = & \mathit{nothing} \\
\mathit{just}\ a \gg= k & = & k\ a \\
\mathit{fail}\ s & = & \mathit{nothing}
\end{array}
\end{array}
$$

Thus, $m \gg= k$ can be seen as a strict postfix application: if m is an exception, the exception is propagated; if m succeeds, then k is applied to the resulting value.

Another well-known application of monads is to model programs that use an internal state. Stateful computations can be represented by functions, so-called *state transformers*, that map an initial state to some value paired with the final state.

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{StateT}\ S\ A & = & \mathit{StateT}\ (S \to (A, S)) \\
\mathit{applyST} & :: & \forall S\ A\,.\, \mathit{StateT}\ S\ A \to S \to (A, S) \\
\mathit{applyST}\ (\mathit{StateT}\ \mathit{st})\ s & = & \mathit{st}\ s \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Monad}\ (\mathit{StateT}\ S)\ \textbf{where}} \\
\quad \mathit{return}\ a & = & \mathit{StateT}\ (\lambda s \to (a, s)) \\
\quad m \gg= k & = & \mathit{StateT}\ (\lambda s \to \textbf{let}\ (a, s') = \mathit{applyST}\ m\ s\ \textbf{in}\ \mathit{applyST}\ (k\ a)\ s')
\end{array}
$$

We will apply state monads in Section 5.2.2.

Despite appearances, *Functor* and *Monad* are closely related. Though this is not reflected in the class declarations, every monad is also a functor. The following definition shows how to define the mapping function in terms of bind and *return*.

$$
\begin{array}{lcl}
\mathit{mmap} & :: & \forall M\ A\ B\,.\,(\mathit{Monad}\ M) \Rightarrow (A \to B) \to (M\ A \to M\ B) \\
\mathit{mmap}\ f\ m & = & m \gg= (\mathit{return} \cdot f)
\end{array}
$$

So, $\mathit{mmap}\ f\ m$ applies f to the result of the computation m.

A *procedure* is a function of type $A \to M\ B$ that maps values to computations. The following operator, called *monadic composition*, composes two procedures. Contrary to the usual composition it also takes care of computational effects.

$$\begin{array}{lcl} (\diamond) & :: & \forall M\ A\ B\ C\,.\,(\mathit{Monad}\ M) \Rightarrow (A \to M\ B) \to (B \to M\ C) \to (A \to M\ C) \\ m_1 \diamond m_2 & = & \lambda a \to m_1\ a \gg\!= m_2 \end{array}$$

The monad laws are easier to remember if we rephrase them in terms of the monadic composition:

$$\begin{array}{lcll} \mathit{return} \diamond k & = & k & \text{(monad law)} \\ k \diamond \mathit{return} & = & k & (\text{——} \shortparallel \text{——}) \\ (k_1 \diamond k_1) \diamond k_2 & = & k_1 \diamond (k_1 \diamond k_2). & (\text{——} \shortparallel \text{——}) \end{array}$$

2.3 Category theory

We have seen in the introduction that generic programs are defined by giving cases for the unit type '1', for sums '+' and for products '×' (and possibly for additional type constructors such as *Char* or *Int*). This section provides a more abstract account of these type constructors. In particular, we will introduce a set of combinators and associated laws that has proven its worth in functional programming, reasoning and program derivation.

The structuring principles underlying the combinators are taken from a branch of mathematics known as category theory. Broadly speaking, category theory is concerned with the study of algebraic structure. The following overview, which summarizes the main definitions, has been compiled from a number of sources, most notably, (Poigné 1992; Bird and de Moor 1997; Backhouse, Jansson, Jeuring, and Meertens 1999; Taylor 1999). The following treatment is rather dense. For a more leisurely exposition the reader is referred to the textbook by Bird and de Moor (1997) or to the excellent tutorial on generic programming by Backhouse, Jansson, Jeuring, and Meertens (1999).

2.3.1 Categories, functors and natural transformations

A *category* $\mathbb{C}$ consists of a class of *objects* and a class of *arrows*. Every arrow f is assigned two objects, a source and a target, written $f : A \to B$. For each object A there is an identity arrow $id_A : A \to A$ and for each pair of arrows $f : A \to B$ and $g : B \to C$ there is a composed arrow $g \cdot f : A \to C$. Identity and composition must satisfy $f \cdot id_A = f = id_A \cdot f$ and $(f \cdot g) \cdot h = f \cdot (g \cdot h)$. The *opposite category* of $\mathbb{C}$, denoted $\mathbb{C}^{op}$, has the same objects and arrows as C, but the source and the target of each arrow are interchanged.

The syntax of a functional programming language such as Haskell can be seen as a category where the objects are types (or rather, equivalence classes of types) and the arrows are terms of the appropriate types (or rather, equivalence classes of terms). Identity and composition are then given by

$$\begin{array}{lcl} id & :: & \forall A\,.\,A \to A \\ id\ a & = & a \\ (\cdot) & :: & \forall A\ B\ C\,.\,(B \to C) \to (A \to B) \to (A \to C) \\ (f \cdot g)\ a & = & f\ (g\ a). \end{array}$$

Other examples of categories are $\mathcal{S}et$, the category of sets and total functions, or $\mathcal{C}po$, the category of complete partial orders and continuous functions, see Section 2.3.8.

A *functor* $F : \mathbb{C} \to \mathbb{D}$ is a structure-preserving mapping between categories. It consists of an object part that maps objects of $\mathbb{C}$ to objects of $\mathbb{D}$ and an arrow part that maps arrows of $\mathbb{C}$ to arrows of $\mathbb{D}$ such that $F\ id = id$ and $F\ (f \cdot g) = F\ f \cdot F\ g$. A functor $F : \mathbb{C}^{op} \to \mathbb{D}$ or $F : \mathbb{C} \to \mathbb{D}^{op}$ is called a *contravariant* functor from $\mathbb{C}$ to $\mathbb{D}$ (the usual case being styled *covariant*). In Haskell, a functor is given by a unary type constructor $F :: \star \to \star$ and an associated mapping function $mapF :: \forall A\ B\,.\,(A \to B) \to (F\ A \to F\ B)$.

A *natural transformation* $\alpha : F \to G : \mathbb{C} \to \mathbb{D}$ is a mapping between functors $F, G : \mathbb{C} \to \mathbb{D}$. It assigns an arrow $\alpha_A :: F\ A \to G\ A$, called a *component*, to each object A of $\mathbb{C}$ such that

$$G\ h \cdot \alpha_A \quad = \quad \alpha_B \cdot F\ h,$$

for all $h : A \to B$. This property is called the *naturality condition*. In Haskell, a natural transformation is a polymorphic function. For instance, the polymorphic function $wrap :: \forall A\,.\,A \to [A]$ can be seen as a natural transformation between Id and $[\,]$ (Haskell's notation for $List$). The associated naturality condition is $map\ h \cdot wrap = wrap \cdot h$ (where map is the mapping function of $[\,]$).

2.3.2 Initial objects

An object '0' is called *initial* if for each object A there is exactly one arrow, written $¡_A$, of type $0 \to A$. The uniqueness of $¡_A$ can be expressed as the following equivalence

$$h = ¡_A \quad \equiv \quad h : 0 \to A,$$

which is known as the *universal property* of '0'. In $\mathcal{S}et$ the initial object is the empty set. In Haskell '0' can be defined using a nullary sum (at least this was possible in Haskell 1.4; Haskell 98 abolished nullary sums):

$$\begin{array}{lcl} \textbf{data}\ 0 & = & \\ ¡ & :: & \forall A\,.\,0 \to A \\ ¡\ n & = & \textbf{case}\ n\ \textbf{of}\ \{\,\}. \end{array}$$

Remark 2.4 Let us assume for the moment that the denotations of Haskell types and Haskell functions live in $\mathcal{S}et$ so that the above declaration defines the empty set. Section 2.3.8 is devoted to finding a suitable category for Haskell. □

2.3.3 Terminal objects

An object '1' is called *terminal* if for each object A there is exactly one arrow, written $!_A$, of type $A \to 1$:

$$h = !_A \quad \equiv \quad h : A \to 1.$$

In $\mathcal{S}et$ every one-element set is terminal (as for all universal constructions terminal objects are unique up to unique isomorphism). In Haskell '1' can be defined by

$$\begin{array}{lcl} \textbf{data}\ 1 & = & () \\ ! & :: & \forall A\,.\,A \to 1 \\ !\ a & = & (). \end{array}$$

REMARK 2.5 Initial and terminal objects are examples of *dual* concepts: an object that is initial in the category $\mathbb{C}$ is terminal in the category $\mathbb{C}^{op}$. □

2.3.4 Products

A *product* of two objects A and B consists of an object, written $A \times B$, and two arrows $outl : A \times B \to A$ and $outr : A \times B \to B$. Products are required to satisfy the following universal property: for each pair of arrows $f : C \to A$ and $g : C \to B$ there exists an arrow, written $f \vartriangle g : C \to A \times B$, such that

$$h = f \vartriangle g \quad \equiv \quad outl \cdot h = f \;\cap\; outr \cdot h = g,$$

for all $h : C \to A \times B$. The universal property of products states that $f \vartriangle g$ is the *unique* arrow satisfying the equations on the right. The arrows $outl$ and $outr$ are sometimes called projections and the combinator $(\vartriangle)$ is known as the 'split' operator. In *Set* products are given by pairing.

If a category has products for each pair of objects, '$\times$' can be made into a so-called *bifunctor* whose associated mapping function is given by:

$$f \times g \quad = \quad (f \cdot outl) \vartriangle (g \cdot outr).$$

The bifunctor laws and several other laws are implied by the universal property:

$$\begin{array}{lcll}
outl \cdot (f \vartriangle g) & = & f & (\vartriangle\text{-computation law}) \\
outr \cdot (f \vartriangle g) & = & g & (\text{———} \;''\; \text{———}) \\
outl \cdot (f \times g) & = & f \cdot outl & (\times\text{-computation law}) \\
outr \cdot (f \times g) & = & g \cdot outr & (\text{———} \;''\; \text{———}) \\
outl \vartriangle outr & = & id & (\text{reflection law}) \\
(f \vartriangle g) \cdot h & = & (f \cdot h) \vartriangle (g \cdot h) & (\vartriangle\text{-fusion law}) \\
id \times id & = & id & (\text{bifunctor law}) \\
(f \times g) \cdot (h \times k) & = & (f \cdot h) \times (g \cdot k) & (\text{——} \;''\; \text{——}) \\
(f \times g) \cdot (h \vartriangle k) & = & (f \cdot h) \vartriangle (g \cdot k). & (\times\text{-}\vartriangle\text{-fusion law})
\end{array}$$

In Haskell products can be defined as follows:

$$\begin{array}{lcl}
\mathbf{data}\; A \times B & = & (A, B) \\
outl & :: & \forall A\, B\,.\, A \times B \to A \\
outl\; (a, b) & = & a \\
outr & :: & \forall A\, B\,.\, A \times B \to B \\
outr\; (a, b) & = & b \\
(\vartriangle) & :: & \forall A\, B\, C\,.\, (C \to A) \to (C \to B) \to (C \to A \times B) \\
(f \vartriangle g)\; c & = & (f\; c, g\; c) \\
(\times) & :: & \forall A_1\, A_2\, B_1\, B_2\,.\, (A_1 \to A_2) \to (B_1 \to B_2) \to (A_1 \times B_1 \to A_2 \times B_2) \\
(f \times g)\; (a, b) & = & (f\; a, g\; b).
\end{array}$$

2.3.5 Coproducts

Coproducts are dual to products.

A *coproduct* of two objects A and B consists of an object, written $A + B$, and two arrows $inl : A \to A + B$ and $inr : B \to A + B$. Coproducts are required to satisfy the following universal property: for each pair of arrows $f : A \to C$ and $g : B \to C$ there exists an arrow, written $f \triangledown g : A + B \to C$, such that

$$h = f \triangledown g \quad \equiv \quad h \cdot inl = f \cap h \cdot inr = g,$$

for all $h : A + B \to C$. The universal property of coproducts states that $f \triangledown g$ is the *unique* arrow satisfying the equations on the right. The arrows *inl* and *inr* are sometimes called injections and the combinator $(\triangledown)$ is known as the 'case' or 'junk' operator. In *Set* coproducts are given by disjoint unions.

If a category has coproducts for each pair of objects, '+' can also be made into a *bifunctor* whose associated mapping function is:

$$f + g \quad = \quad (inl \cdot f) \triangledown (inr \cdot g).$$

The universal property implies the bifunctor laws and several other laws:

$$\begin{array}{lcll}
(f \triangledown g) \cdot inl & = & f & (\triangledown\text{-computation law}) \\
(f \triangledown g) \cdot inr & = & g & (\text{———— " ————}) \\
(f + g) \cdot inl & = & inl \cdot f & (+\text{-computation law}) \\
(f + g) \cdot inr & = & inr \cdot g & (\text{———— " ————}) \\
inl \triangledown inr & = & id & (\text{reflection law}) \\
h \cdot (f \triangledown g) & = & (h \cdot f) \triangledown (h \cdot g) & (\triangledown\text{-fusion law}) \\
id + id & = & id & (\text{bifunctor law}) \\
(f + g) \cdot (h + k) & = & (f \cdot h) + (g \cdot k) & (\text{—— " ——}) \\
(f \triangledown g) \cdot (h + k) & = & (f \cdot h) \triangledown (g \cdot k) & (\triangledown\text{-+-fusion law})
\end{array}$$

In Haskell coproducts can be defined as follows:

$$\begin{array}{lcl}
\mathbf{data}\ A + B & = & inl\ A \mid inr\ B \\
(\triangledown) & :: & \forall A\ B\ C\,.\,(A \to C) \to (B \to C) \to (A + B \to C) \\
(f \triangledown g)\ (inl\ a) & = & f\ a \\
(f \triangledown g)\ (inr\ b) & = & g\ b \\
(+) & :: & \forall A_1\ A_2\ B_1\ B_2\,.\,(A_1 \to A_2) \to (B_1 \to B_2) \to (A_1 + B_1 \to A_2 + B_2) \\
(f + g)\ (inl\ a) & = & inl\ (f\ a) \\
(f + g)\ (inr\ b) & = & inr\ (g\ b).
\end{array}$$

2.3.6 Exponentials

Assume that we have a category with a terminal object and products. An *exponent* of two objects A and B is an object, written B^A, and an arrow $eval : B^A \times A \to B$. Exponents are required to satisfy the following universal property: for each arrow $f : A \times B \to C$ there exists an arrow, written $curry\ f : A \to C^B$, such that

$$g = curry\ f \quad \equiv \quad eval \cdot (g \times id) = f,$$

for all $g : A \to C^B$. If a category has a terminal object, products, and for every pair of objects the exponential B^A exists, the category is said to be *cartesian closed*. In *Set*, which is cartesian closed, exponents are sets of total functions.

REMARK 2.6 An alternative characterization of exponentials is based on the isomorphism $A \times B \to C \cong A \to C^B$, which allows us to turn a binary function

$f : A \times B \to C$ into a unary, higher-order function $curry\ f : A \to C^B$ and conversely a function $g : A \to C^B$ into a function $uncurry\ g : A \times B \to C$. The universal property reads:

$$g = curry\ f \quad \equiv \quad uncurry\ g = f.$$

The arrows *eval* and *uncurry* are interdefinable: $uncurry\ g = eval \cdot (g \times id)$ and $eval = uncurry\ id$. □

Contrary to products and coproducts, $(-)^{(-)}$ cannot be made into a bifunctor. Rather, $(-)^{(-)}$ serves as an example of a so-called *difunctor*. A difunctor is *contravariant* in its first argument and *covariant* in its second. Its mapping function is given by

$$g^f \quad = \quad curry\ (g \cdot eval \cdot (id \times f)).$$

As usual, the universal property implies the difunctor laws and several others:

$$
\begin{array}{lcll}
eval \cdot (curry\ f \times id) & = & f & \text{(computation law)} \\
curry\ (eval \cdot (g \times id)) & = & g & \text{(——— " ———)} \\
curry\ eval & = & id & \text{(reflection law)} \\
curry\ f \cdot g & = & curry\ (f \cdot (g \times id)) & \text{(fusion law)} \\
id^{id} & = & id & \text{(difunctor law)} \\
g^f \cdot k^h & = & (g \cdot k)^{(h \cdot f)} & \text{(—— " ——)}
\end{array}
$$

In Haskell exponents are simply function spaces.

$$
\begin{array}{lcl}
\textbf{type}\ B^A & = & A \to B \\
curry & :: & \forall A\ B\ C\,.\,(A \times B \to C) \to (A \to C^B) \\
curry\ f\ a\ b & = & f\ (a, b) \\
uncurry & :: & \forall A\ B\ C\,.\,(A \to C^B) \to (A \times B \to C) \\
uncurry\ g\ (a, b) & = & g\ a\ b \\
eval & :: & \forall A\ B\,.\,B^A \times A \to B \\
eval\ (f, a) & = & f\ a \\
(-)^{(-)} & :: & \forall A_1\ A_2\ B_1\ B_2\,.\,(A_2 \to A_1) \to (B_1 \to B_2) \to (B_1^{A_1} \to B_2^{A_2}) \\
g^f & = & \lambda h \to g \cdot h \cdot f
\end{array}
$$

Note that the pointwise definition of the mapping function is much simpler than the point-free definition in terms of *eval* and *curry*.

2.3.7 Isomorphisms

An *isomorphism* is an arrow $i : A \to B$ that has an inverse, written $i^{-1} : B \to A$, such that $i^{-1} \cdot i = id_A$ and $i \cdot i^{-1} = id_B$. If there exists an isomorphism $i : A \to B$, A and B are said to be *isomorphic*, and we write $i : A \cong B : i^{-1}$ or simply $A \cong B$.

A natural transformation is called a *natural isomorphism* if its components are isomorphisms. For instance, the natural transformation $swap : A \times B \to B \times A = outr \vartriangle outl$ is an isomorphism with associated naturality property:

$$(g \times f) \cdot swap \quad = \quad swap \cdot (f \times g).$$

In every category with a terminal object and products there exist the following natural isomorphisms:

$$\begin{array}{rclcll} \mathit{unit} & : & 1 \times A & \cong & A & : & \mathit{ununit} \\ \mathit{swap} & : & A \times B & \cong & B \times A & : & \mathit{swap} \\ \mathit{assocl} & : & A \times (B \times C) & \cong & (A \times B) \times C & : & \mathit{assocr}. \end{array}$$

Dually, in every category with an initial object and coproducts there exist the following natural isomorphisms:

$$\begin{array}{rclcll} \mathit{zero} & : & 0 + A & \cong & A & : & \mathit{unzero} \\ \mathit{mirror} & : & A + B & \cong & B + A & : & \mathit{mirror} \\ & & A + (B + C) & \cong & (A + B) + C. & & \end{array}$$

A category with products and coproducts is called *distributive* if there exist natural isomorphisms:

$$\begin{array}{rclcll} & & 0 \times A & \cong & 0 & & \\ \mathit{distl} & : & (B + C) \times A & \cong & (B \times A) + (C \times A) & : & \mathit{undistl}. \end{array}$$

The function *distl* distributes '$\times$' leftward through '$+$'. Note that any cartesian closed category that has coproducts is distributive.

Finally, in a cartesian closed category there exist natural isomorphisms:

$$\begin{array}{rcl} A^0 & \cong & 1 \\ A^1 & \cong & A \\ A^{B+C} & \cong & A^B \times A^C \\ A^{B \times C} & \cong & (A^B)^C. \end{array}$$

These isomorphisms are also known as the *laws of exponentials*.

Here are some of the isomorphisms programmed in Haskell.

$$\begin{array}{lcl} \mathit{unit} & :: & \forall A \, . \, 1 \times A \to A \\ \mathit{unit} \; ((), a) & = & a \\ \mathit{ununit} & :: & \forall A \, . \, A \to 1 \times A \\ \mathit{ununit} \; a & = & ((), a) \\ \mathit{swap} & :: & \forall A \; B \, . \, A \times B \to B \times A \\ \mathit{swap} \; (a, b) & = & (b, a) \\ \mathit{assocl} & :: & \forall A \; B \; C \, . \, A \times (B \times C) \to (A \times B) \times C \\ \mathit{assocl} \; (a, (b, c)) & = & ((a, b), c) \\ \mathit{assocr} & :: & \forall A \; B \; C \, . \, (A \times B) \times C \to A \times (B \times C) \\ \mathit{assocr} \; ((a, b), c) & = & (a, (b, c)) \\ \mathit{distl} & :: & \forall A \; B \; C \, . \, (A + B) \times C \to (A \times C) + (B \times C) \\ \mathit{distl} \; (\mathit{inl} \; a, c) & = & \mathit{inl} \; (a, c) \\ \mathit{distl} \; (\mathit{inr} \; b, c) & = & \mathit{inr} \; (b, c) \\ \mathit{undistl} & :: & \forall A \; B \; C \, . \, (A \times C) + (B \times C) \to (A + B) \times C \\ \mathit{undistl} \; (\mathit{inl} \; (a, c)) & = & (\mathit{inl} \; a, c) \\ \mathit{undistl} \; (\mathit{inr} \; (b, c)) & = & (\mathit{inr} \; b, c) \end{array}$$

2.3.8 Fixed points

In the previous sections we have shown how to implement each of the categorical constructions in Haskell. We tacitly assumed that we are working in the category

Set, the category of sets and total functions. Unfortunately, full Haskell cannot be given a semantics in *Set* since Haskell provides unbounded recursion. It is a lamentable fact that cartesian closure, coproducts and fixed points cannot coexist, see (Huwig and Poigné 1990). A category has fixed points if for every object A there is a fixed point combinator $\mathit{fix}_A : A^A \to A$. A cartesian-closed category that has coproducts and fixed points is a preorder, that is, $A \cong 1$ for every object A.

The usual resort is to work with complete partial orders and continuous functions instead of sets and total functions. Recall that a partially ordered set is *complete* if every directed subset has a least upper bound; it is *pointed* if it has a least element. A function is *continuous* if it preserves least upper bounds; it is *strict* if it preserves the least element. Let D be a complete, pointed partial order and let $\varphi : \mathsf{D} \to \mathsf{D}$ be a continuous function. The fixed point theorem then shows that $\bigsqcup\{\varphi^n(\bot) \mid n \in \mathbb{N}\}$ is the *least fixed point* of φ.

Now, sacrificing one of the three properties 'cartesian closure', 'has coproducts', or 'has fixed points' we obtain one of the following three categories:

Cpo, the category of complete partial orders and continuous functions: it has categorical products (the cartesian product '$\times$'); it is cartesian closed; it has categorical coproducts (the disjoint union '$\uplus$'); $\emptyset$ is the initial object; $\{\bot\}$ is the terminal object; it has fixed points for every pointed object (however, if D is not pointed, then a continuous function $\varphi : \mathsf{D} \to \mathsf{D}$ may not have a fixed point).

Cppo, the category of complete, pointed partial orders and continuous functions: it has categorical products (the cartesian product '$\times$'); it is cartesian closed; it has no coproducts and no initial object; $\{\bot\}$ is the terminal object; it has fixed points on every object.

$\mathit{Cpo}_\bot$, the category of complete, pointed partial orders and strict, continuous functions: it has products (the cartesian product '$\times$'), but it is not cartesian closed; it is, however, monoidally closed (the smash product '$\otimes$' and the space '$\multimap$' of strict continuous functions form a monoidal closure[2]); it has categorical coproducts (the coalesced sum '$\oplus$', see below); $\{\bot\}$ is both initial and terminal.

2.3.9 A semantics for data declarations

Each of the three categories listed in the previous section can be used to give a semantics for Haskell. For instance, Launchbury and Paterson (1996) show how to interpret Haskell in *Cpo* by restricting the fixed point operator to pointed objects. The last category, $\mathit{Cpo}_\bot$, is particularly attractive, since it allows to define a precise semantics for Haskell's **data** construct (including strictness annotations). To this end let us introduce three constructions on partially ordered sets: *lifts*, *coalesced sums* (or amalgated sums) and *smash products* (or strict products):

$$\begin{aligned}
\mathsf{D}_\bot &= \{(0,\delta) \mid \delta\} \cup \{\bot\} \\
\mathsf{D} \oplus \mathsf{E} &= \{(0,\delta) \mid \delta \in \mathsf{D} \setminus \{\bot\}\} \cup \{(1,\epsilon) \mid \epsilon \in \mathsf{E} \setminus \{\bot\}\} \cup \{\bot\} \\
\mathsf{D} \otimes \mathsf{E} &= \{(\delta,\epsilon) \mid \delta \in \mathsf{D} \setminus \{\bot\}, \epsilon \in \mathsf{E} \setminus \{\bot\}\} \cup \{\bot\}.
\end{aligned}$$

Given these constructions the right-hand side of the data type declaration

$$\mathbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}$$

is interpreted by

$$((D_{11})_\bot \otimes \cdots \otimes (D_{1m_1})_\bot) \oplus \cdots \oplus ((D_{n1})_\bot \otimes \cdots \otimes (D_{nm_n})_\bot)$$

[2] Monoidal closure is similar to cartesian closure except that the product (here, the smash product) is not a categorical product but a *tensor product* (MacLane 1998).

where D_{ij} is the interpretation of T_{ij} (if the type T_{ij} has a strictness flag '!', then D_{ij} is not lifted). Since $0_\perp \oplus D \cong D$ and $1_\perp \otimes D \cong D$ (where $0 = \emptyset$ and $1 = \{\perp\}$), we set $D_1 \oplus \cdots \oplus D_n = 0_\perp$ and $D_1 \otimes \cdots \otimes D_n = 1_\perp$ for $n = 0$. Consequently, the data type declarations (corresponding to '0', '1', '+', '×' and '⊗')

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Void} & & \\
\mathbf{data}\ () & = & () \\
\mathbf{data}\ \mathit{Either}\ A\ B & = & \mathit{left}\ A \mid \mathit{right}\ B \\
\mathbf{data}\ (A, B) & = & (A, B) \\
\mathbf{data}\ \mathit{Smash}\ A\ B & = & \mathit{smash}\ !\ A\ !\ B
\end{array}
$$

are interpreted by (mixing syntax and semantics)

$$
\begin{array}{lcl}
\mathit{Void} & = & 0_\perp \\
() & = & 1_\perp \\
\mathit{Either}\ A\ B & = & A_\perp \oplus B_\perp \\
(A, B) & = & A_\perp \otimes B_\perp \\
\mathit{Smash}\ A\ B & = & A \otimes B.
\end{array}
$$

Note that $A_\perp \oplus B_\perp$ is isomorphic to the so-called *separated sum* (usually written '+') and that $A_\perp \otimes B_\perp \cong (A \times B)_\perp$ is a *lifted product.*

Not only sums and products are lifted in Haskell, but also functional types, that is, $T \to U$ is interpreted by $[D \to E]_\perp \cong [D_\perp \multimap E]_\perp$ where D is the interpretation of T and E is the interpretation of U. Unfortunately, this implies that η-conversion is not valid since $\lambda a\,.\,\perp = \lambda a\,.\,\perp a \neq \perp$.

The bottom line of all this is that almost none of the laws we have seen so far holds in Haskell. Or, to put it positively, most of the laws are subject to side conditions. For instance,

$$h = f \triangledown g \quad \equiv \quad h \cdot \mathit{inl} = f \cap h \cdot \mathit{inr} = g$$

holds only for *strict* $h : A + B \to C$.

Somewhat ironically, even $\mathit{uncurry}\ (\mathit{curry}\ f) = f$ does not hold in Haskell[3], since Haskell has lifted products.

2.4 The simply typed λ-calculus

This section deals with the simply typed λ-calculus, its syntax and semantics. Essentially, it prepares the ground for the next section which is dedicated to the polymorphic λ-calculus. Besides, we will introduce the main proof technique used in this thesis, which is based on so-called logical relations. For a more leisurely exposition the reader is referred to the excellent textbook by Mitchell (1996).

[3] Recall that Haskell is named after the logician Haskell B. Curry.

$$\frac{}{C :: \star}\ (\textsc{t-const-form})$$

$$\frac{T :: \star \qquad U :: \star}{(T \times U) :: \star}\ (\times\textsc{-form}) \qquad \frac{T :: \star \qquad U :: \star}{(T \to U) :: \star}\ (\to\textsc{-form})$$

Figure 2.1: Type formation rules.

2.4.1 Syntax

Syntactic categories The simply typed λ-calculus has a two-level structure.

type constants	$C, D \in \mathit{Const}$
type terms	$T, U \in \mathit{Type}$
individual constants	$c, d \in \mathit{const}$
individual variables	$a, b \in \mathit{var}$
terms	$t, u \in \mathit{term}$

We use upper-case Roman letters for types and lower-case Roman letters for terms.

Types Type terms are formed according to the following grammar.

$$\begin{array}{rcll} T, U \in \mathit{Type} & ::= & C & \text{type constant} \\ & \mid & T \times U & \text{product type} \\ & \mid & T \to U & \text{function type} \end{array}$$

We agree upon the convention that '$\times$' and '$\to$' associate to the right, that is, $T_1 \to T_2 \to T_3$ stands for $T_1 \to (T_2 \to T_3)$.

The construction of type terms can alternatively be formalized using so-called *type formation rules*, see Figure 2.1. Here, '$\star$' denotes the 'type' of types.

Terms Terms are built from constants and variables using pairing, projection, abstraction, application and recursion. It is convenient to assume that constants and variables are typed, that is, a constant or a variable is a pair consisting of a name and a type usually written $s :: T$. If $s :: T$ is a typed constant or a typed variable, we define 'type $(s :: T) = T$'. Furthermore, we assume that for each type there is an infinite number of typed variables (we could set, for instance, $\mathit{var} = \Sigma^* \times \mathit{Type}$ where Σ is some non-empty alphabet). *Pseudo-terms* (also called *raw terms*) are formed according to the following grammar.

$$\begin{array}{rcll} t, u \in \mathit{term} & ::= & c & \text{constant} \\ & \mid & a & \text{variable} \\ & \mid & (t_1, t_2) & \text{pairing} \\ & \mid & \mathit{outl}\ t & \text{projection} \\ & \mid & \mathit{outr}\ t & \text{projection} \\ & \mid & \lambda a\,.\,t & \text{abstraction} \\ & \mid & t\ u & \text{application} \\ & \mid & \mathit{fix}\ t & \text{recursion} \end{array}$$

Here, (t_1, t_2) denotes pairing, $\mathit{outl}\ t$ projects onto the first component of t, $\mathit{outr}\ t$ projects onto the second component, $\lambda a\,.\,t$ denotes abstraction, $t\ u$ denotes application, and $\mathit{fix}\ t$ denotes the fixed point of t. We assume that application

$$\frac{}{c :: \text{type } c}\ (\textsc{const}) \qquad \frac{}{a :: \text{type } a}\ (\textsc{var})$$

$$\frac{t_1 :: T_1 \qquad t_2 :: T_2}{(t_1, t_2) :: (T_1 \times T_2)}\ (\times\textsc{-intro})$$

$$\frac{t :: (T_1 \times T_2)}{(\mathit{outl}\ t) :: T_1}\ (\times\textsc{-elim-l}) \qquad \frac{t :: (T_1 \times T_2)}{(\mathit{outr}\ t) :: T_2}\ (\times\textsc{-elim-r})$$

$$\frac{t :: T}{(\lambda a \,.\, t) :: (\text{type } a \to T)}\ (\to\textsc{-intro}) \qquad \frac{t :: (U \to V) \qquad u :: U}{(t\ u) :: V}\ (\to\textsc{-elim})$$

$$\frac{t :: (U \to U)}{(\mathit{fix}\ t) :: U}\ (\textsc{fix})$$

Figure 2.2: Typing rules.

associates to the left and that abstraction extends as far to the right as possible. Finally, we abbreviate nested abstractions $\lambda a_1 \,.\, \ldots \lambda a_m \,.\, t$ by $\lambda a_1\ \ldots\ a_m \,.\, t$.

Remark 2.7 The fairly standard syntax for abstraction is different from Haskell syntax: $\lambda a \,.\, t$ is written $\lambda a \to t$ in Haskell. Keep this in mind when reading the examples in later chapters, which usually employ Haskell syntax. □

Pseudo-terms are syntactically well-formed but they may be ill-typed. Consider, for instance, the pseudo-term $c\ c$ where c is some constant of type C. A pseudo-term t is called a *term* if there is some type T such that $t :: T$ is derivable using the *typing rules* depicted in Figure 2.2. It is worth noting that since constants and variables are annotated with their types, we do not require an explicit typing environment.

The axiomatic semantics of the simply typed λ-calculus is given by the convertibility relation.

Definition 2.8 The *convertibility relation*, denoted '$\leftrightarrow$', is given by the following axioms

$$\begin{array}{llll}
\mathit{outl}\ (t_1, t_2) & \leftrightarrow & t_1 & & (\pi_1) \\
\mathit{outr}\ (t_1, t_2) & \leftrightarrow & t_2 & & (\pi_1) \\
(\mathit{outl}\ t, \mathit{outr}\ t) & \leftrightarrow & t & & (\pi) \\
(\lambda a \,.\, t)\ u & \leftrightarrow & t[\,a := u\,] & & (\beta) \\
\lambda a \,.\, t\ a & \leftrightarrow & t & a \text{ not free in } t & (\eta) \\
\mathit{fix}\ t & \leftrightarrow & t\ (\mathit{fix}\ t) & & (\mu)
\end{array}$$

plus rules for reflexivity, symmetry, transitivity and congruence. □

2.4.2 Semantics

This section is concerned with the denotational semantics of the simply typed λ-calculus. There are two general frameworks for describing the semantics: environment models and models based on cartesian closed categories. We will use *environment models* for the presentation since they are somewhat easier to understand. Since the term language includes a fixed point operator, we will furthermore restrict ourselves to domain-theoretic interpretations, where a domain is an algebraic semilattice—a complete partial order with some additional properties, see (Gunter and Scott 1990). If D and E are domains, then $[\mathsf{D} \to \mathsf{E}]$ denotes the set of all continuous functions from D to E.

The definition of the semantics proceeds in three steps. First, we introduce so-called applicative structures, and then we define two conditions that an applicative structure must satisfy to qualify as a model.

Definition 2.9 An *applicative structure* $\mathcal{E}$ is a tuple $(\mathsf{E}, \mathbf{outl}, \mathbf{outr}, \mathbf{app}, \mathbf{const})$ such that

- $\mathsf{E} = (\mathsf{E}^T \mid T \in \mathit{Type})$ is a family of domains,
- $\mathbf{outl} = (\mathbf{outl}_{T,U} \mid T, U \in \mathit{Type})$ and $\mathbf{outr} = (\mathbf{outr}_{T,U} \mid T, U \in \mathit{Type})$ are families of continuous functions with $\mathbf{outl}_{T,U} : [\mathsf{E}^{T\times U} \to \mathsf{E}^T]$ and $\mathbf{outr}_{T,U} : [\mathsf{E}^{T\times U} \to \mathsf{E}^U]$, and
- $\mathbf{app} = (\mathbf{app}_{T,U} \mid T, U \in \mathit{Type})$ is a family of continuous functions with $\mathbf{app}_{T,U} : [\mathsf{E}^{T\to U} \to [\mathsf{E}^T \to \mathsf{E}^U]]$, and
- $\mathbf{const} : \mathit{const} \to \mathsf{E}$ is a mapping from individual constants to values such that $\mathbf{const}(c) \in \mathsf{E}^T$ for all $c \in \mathit{const}$ with $T = \text{type } c$.

A *type frame* is an applicative structure such that

- $\mathsf{E}^{T\times U} \subseteq \mathsf{E}^T \times \mathsf{E}^U$, $\mathbf{outl}_{T,U}\ (\delta, \epsilon) = \delta$ and $\mathbf{outr}_{T,U}\ (\delta, \epsilon) = \epsilon$, and
- $\mathsf{E}^{T\to U} \subseteq \mathsf{E}^T \to \mathsf{E}^U$ and $\mathbf{app}_{T,U}\ \varphi\ \delta = \varphi(\delta)$. □

The first condition on models requires that two elements representing pairs are equal if they have the same components and that equality between elements of function types is standard equality on functions.

Definition 2.10 An applicative structure $\mathcal{E} = (\mathsf{E}, \mathbf{outl}, \mathbf{outr}, \mathbf{app}, \mathbf{const})$ is *extensional*, if

- $\forall \pi_1, \pi_2 \in \mathsf{E}^{T\times U} . (\mathbf{outl}\ \pi_1 = \mathbf{outl}\ \pi_2 \cap \mathbf{outr}\ \pi_1 = \mathbf{outr}\ \pi_2) \supset \pi_1 = \pi_2$, and
- $\forall \varphi_1, \varphi_2 \in \mathsf{E}^{T\to U} . (\forall \delta \in \mathsf{E}^T . \mathbf{app}\ \varphi_1\ \delta = \mathbf{app}\ \varphi_2\ \delta) \supset \varphi_1 = \varphi_2$. □

The second condition on models ensures that the applicative structure has enough points so that every term containing pairs and λ-abstractions can be assigned a meaning in the structure. To formulate the condition we require the notion of an *environment*. An environment η is a mapping $\eta : \mathit{var} \to \mathsf{E}$ such that $\eta(a) \in \mathsf{E}^T$ for all $a \in \mathit{var}$ with $T = \text{type } a$. If η is an environment, then $\eta(a := \delta)$ is the environment mapping a to δ and b to $\eta(b)$ for b different from a.

Definition 2.11 An applicative structure $\mathcal{E} = (\mathsf{E}, \mathbf{outl}, \mathbf{outr}, \mathbf{app}, \mathbf{const})$ satisfies the *environment model condition* if the clauses below define a total meaning function, where the meaning function is defined by induction on the structure of typing derivations.

$$
\begin{array}{lcl}
\mathcal{E}[\![t :: T]\!]\eta & \in & \mathbf{E}^{T} \\
\mathcal{E}[\![c :: T]\!]\eta & = & \mathbf{const}(c) \\
\mathcal{E}[\![a :: T]\!]\eta & = & \eta(a) \\
\mathcal{E}[\![(t_1, t_2) :: (T_1 \times T_2)]\!]\eta & = & \text{the unique } \pi \in \mathbf{E}^{T_1 \times T_2} \text{ such that} \\
 & & \quad \mathbf{outl}_{T_1,T_2}\ \pi = \mathcal{E}[\![t_1 :: T_1]\!]\eta \text{ and} \\
 & & \quad \mathbf{outr}_{T_1,T_2}\ \pi = \mathcal{E}[\![t_2 :: T_2]\!]\eta \\
\mathcal{E}[\![(\mathit{outl}\ t) :: T_1]\!]\eta & = & \mathbf{outl}_{T_1,T_2}\ (\mathcal{E}[\![t :: T_1 \times T_2]\!]\eta) \\
\mathcal{E}[\![(\mathit{outr}\ t) :: T_2]\!]\eta & = & \mathbf{outr}_{T_1,T_2}\ (\mathcal{E}[\![t :: T_1 \times T_2]\!]\eta) \\
\mathcal{E}[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta & = & \text{the unique } \varphi \in \mathbf{E}^{S \to T} \text{ such that} \\
 & & \quad \forall \delta \in \mathbf{E}^{S} \,.\, \mathbf{app}_{S,T}\ \varphi\ \delta = \mathcal{E}[\![t :: T]\!]\eta(a := \delta) \\
\mathcal{E}[\![(t\ u) :: V]\!]\eta & = & \mathbf{app}_{U,V}\ (\mathcal{E}[\![t :: U \to V]\!]\eta)\ (\mathcal{E}[\![u :: U]\!]\eta) \\
\mathcal{E}[\![(\mathit{fix}\ t) :: U]\!]\eta & = & \bigsqcup\{\delta_n \mid n \in \mathbb{N}\} \\
 & & \text{where } \delta_0 \;=\; \bot \\
 & & \qquad\quad \delta_{n+1} \;=\; \mathbf{app}_{U,U}\ (\mathcal{E}[\![t :: U \to U]\!]\eta)\ \delta_n
\end{array}
$$

An extensional, applicative structure that satisfies the environment model condition is called an *environment model for the simply typed λ-calculus*. □

Note that extensionality guarantees the uniqueness of the elements π and φ whose existence is postulated in the third and in the sixth clause, respectively. So an extensional, applicative structure might only fail to satisfy the environment model condition if $\mathbf{E}^{T_1 \times T_2}$ or $\mathbf{E}^{S \to T}$ does not contain enough elements. As an aside, the clause for $\mathcal{E}[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta$ can be written more succinctly using meta abstraction and inverse application:

$$\mathcal{E}[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta \quad = \quad \mathbf{app}^{-1}_{S,T}\ (\boldsymbol{\lambda}\delta \in \mathbf{E}^{S} \,.\, \mathcal{E}[\![t :: T]\!]\eta(a := \delta)).$$

We will sometimes use this notation as it is more compact.

The following fact states that the axiomatic semantics is sound with respect to the denotational semantics.

FACT 2.12 Let $\mathcal{E}$ be a model and let t_1 and t_2 be two terms of type T, then

$$t_1 \leftrightarrow t_2 \quad \supset \quad \forall \eta \,.\, \mathcal{E}[\![t_1]\!]\eta = \mathcal{E}[\![t_2]\!]\eta. \qquad \square$$

The environment model condition is often difficult to check. An equivalent, but simpler condition is the combinatory model condition.

DEFINITION 2.13 An applicative structure $\mathcal{E} = (\mathbf{E}, \mathbf{outl}, \mathbf{outr}, \mathbf{app}, \mathbf{const})$ satisfies the *combinatory model condition* if

- for all types T and U there exist elements $\mathbf{P} \in \mathbf{E}^{T \to U \to (T \times U)}$, $\mathbf{L} \in \mathbf{E}^{(T \times U) \to T}$ and $\mathbf{R} \in \mathbf{E}^{(T \times U) \to U}$ such that

$$
\begin{array}{lcl}
\mathbf{app}\ \mathbf{L}\ (\mathbf{app}\ (\mathbf{app}\ \mathbf{P}\ x)\ y) & = & x \\
\mathbf{app}\ \mathbf{R}\ (\mathbf{app}\ (\mathbf{app}\ \mathbf{P}\ x)\ y) & = & y \\
\mathbf{app}\ (\mathbf{app}\ \mathbf{P}\ (\mathbf{app}\ \mathbf{L}\ z))\ (\mathbf{app}\ \mathbf{R}\ z) & = & z
\end{array}
$$

  for all x, y and z of the appropriate types.

- for all types S, T and U there exist elements $\mathsf{K} \in \mathsf{E}^{T \to U \to T}$ and $\mathsf{S} \in \mathsf{E}^{(T \to U \to V) \to (T \to U) \to T \to V}$ such that

$$\begin{array}{lcl} \mathsf{app}\ (\mathsf{app}\ \mathsf{K}\ x)\ y & = & x \\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ \mathsf{S}\ x)\ y)\ z & = & \mathsf{app}\ (\mathsf{app}\ x\ z)\ (\mathsf{app}\ y\ z) \end{array}$$

for all x, y and z of the appropriate types. □

2.4.3 Böhm trees

The simply typed λ-calculus can be interpreted in a syntactic way using so-called Böhm trees. One can think of Böhm trees as a kind of 'infinite normal form' for λ-terms, which is obtained by unwinding a λ-term ad infinitum.

Definition 2.14 A *head-normal form* is a term of the form $\lambda a_1\ \ldots\ a_m\,.\,z\ t_1\ \ldots\ t_n$ with $m, n \geqslant 0$ and $z \in \mathit{const} \cup \mathit{var}$. A term t has head-normal form u if $t \leftrightarrow u$ and u is a head-normal form. □

Definition 2.15 A head-normal form $\lambda a_1\ \ldots\ a_m\,.\,z\ t_1\ \ldots\ t_n$ of type $T_1 \to \cdots \to T_{m'} \to C$ is a *η-head-normal form* if $m = m'$. A term t has η-head-normal form u if $t \leftrightarrow u$ and u is a η-head-normal form. □

Not every term has an η-head-normal form, consider, for instance, $\mathit{fix}\ (\lambda a\,.\,a) \leftrightarrow (\lambda a\,.\,a)\ (\mathit{fix}\ (\lambda a\,.\,a))$. Contrary to the untyped λ-calculus, however, it is decidable whether a term possesses an η-head-normal form. For that reason the notion of Böhm tree introduced below is effective.

Definition 2.16 The *Böhm tree* of the term t, denoted $\mathrm{BT}(t)$, is a labelled, possibly infinite tree defined as follows: if the term t has η-head-normal form $\lambda a_1\ \ldots\ a_m\,.\,z\ t_1\ \ldots\ t_n$, then

$$\mathrm{BT}(t) \;=\; \lambda a_1\ \ldots\ a_m\,.\,\begin{array}{c} z \\ \swarrow \quad \searrow \\ \mathrm{BT}(t_1) \quad \cdots \quad \mathrm{BT}(t_n) \end{array}.$$

Otherwise, if t has no η-head-normal form, then $\mathrm{BT}(t) = \Omega$. A *Böhm-like tree* is a possibly infinite tree labelled with objects of the form $\lambda a_1\ \ldots\ a_m\,.\,z$. The set of all well-typed Böhm-like trees of type T is denoted $\mathcal{B}^T$. □

Example 2.17 We can rewrite the types introduced in Section 2.1 as λ-terms if we view '1', '+' and '×' as constants over some base type, say, *Nat*. The types *List* and *Perfect*, for instance, correspond to

$$\begin{array}{lcl} \mathit{list} & = & \lambda a\,.\,\mathit{fix}\ (\lambda l\,.\,1 + a \times l) \\ \mathit{perfect} & = & \mathit{fix}\ (\lambda p\,.\,\lambda a\,.\,a + p\ (a \times a)). \end{array}$$

The Böhm trees of *list* and *perfect* are displayed in Figure 2.3. Note that *list* yields a *rational tree* while *perfect* gives rise to an *algebraic tree*. A rational tree is a possibly infinite tree that has only a finite number of subtrees. Algebraic trees are obtained as solutions of so-called algebraic equations, see, for instance, (Courcelle 1983). □

Böhm trees induce a congruence relation on λ-terms.

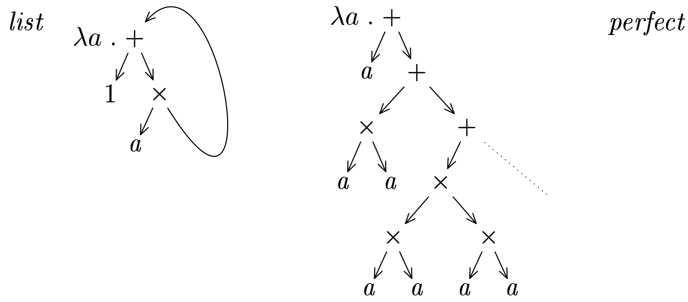


Figure 2.3: The Böhm trees of *list* and *perfect*.

Definition 2.18 Let t_1 and t_2 be two terms of type T. We define

$$t_1 \approx t_2 \quad \equiv \quad \mathrm{BT}(t_1) = \mathrm{BT}(t_2).$$

If $t_1 \approx t_2$, we say t_1 and t_2 are *structurally equivalent*. □

It is in general undecidable whether two λ-terms are related by $(\approx)$. The problem becomes decidable if we restrict *fix* to type constants or to first-order types. In the first case we obtain rational trees, in the latter case we obtain algebraic trees. Though decidable, the equality problem for algebraic trees is non-trivial. It has been known for a long time that this problem and the equivalence problem for deterministic pushdown automata are interreducible (Courcelle 1983). It was, however, only recently shown that the latter problem is decidable (Sénizergues 1997).

The set $\mathcal{B}^T$ of all well-typed Böhm-like trees of type T can be turned into a domain by imposing some suitable partial order. In fact, $\mathcal{B}^T$ gives rise to a model of the simply typed λ-calculus, the so-called *Böhm-tree model*. The details of the construction are quite technical, so we will not repeat them here. Instead, we refer the interested reader to (Barendregt 1984).

2.4.4 Logical relations

Logical relations are an important tool in the study of typed λ-calculi. In fact, most of the proofs in this thesis are based on (variants of) logical relations. For that reason, the reader is urged to study this section in some detail. For a comprehensive treatment of logical relations the reader is referred to Mitchell's textbook (1996).

In presenting logical relations we will restrict ourselves to the binary case. The extension to the n-ary case is, however, entirely straightforward.

Definition 2.19 Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two applicative structures. A logical relation $\mathcal{R} = (\mathcal{R}^T \mid T \in \mathit{Type})$ over $\mathcal{E}_1$ and $\mathcal{E}_2$ is a family of relations such that

- $\mathcal{R}^T \subseteq \mathbf{E}_1^T \times \mathbf{E}_2^T$,
- $(\mathbf{const}_1(c), \mathbf{const}_2(c)) \in \mathcal{R}^T$ for all $c \in \mathit{const}$ with $T = \text{type } c$,
- $\mathcal{R}^{T \times U}$ is closed under pairing and projection:

$$\begin{aligned}&(\pi_1, \pi_2) \in \mathcal{R}^{T \times U}\\ &\qquad \equiv \; (\mathbf{outl}_1\ \pi_1, \mathbf{outl}_2\ \pi_2) \in \mathcal{R}^T \;\cap\; (\mathbf{outr}_1\ \pi_1, \mathbf{outr}_2\ \pi_2) \in \mathcal{R}^U,\end{aligned}$$

- $\mathcal{R}^{T \to U}$ is closed under application and abstraction:

$$\begin{aligned}&(\varphi_1, \varphi_2) \in \mathcal{R}^{T \to U} \\ &\quad \equiv \forall \delta_1 \in \mathbf{E}_1^T, \delta_2 \in \mathbf{E}_2^T \,. \\ &\qquad\qquad (\delta_1, \delta_2) \in \mathcal{R}^T \supset (\mathbf{app}_1\ \varphi_1\ \delta_1, \mathbf{app}_2\ \varphi_2\ \delta_2) \in \mathcal{R}^U,\end{aligned}$$

- $\mathcal{R}^T$ is pointed, that is, $(\bot, \bot) \in \mathcal{R}^T$,
- $\mathcal{R}^T$ is chain-complete, that is, $S \subseteq \mathcal{R}^T \supset \bigsqcup S \in \mathcal{R}^T$ for every chain S. □

Usually, a logical relation is defined on type constants only; the third clause of the definition then shows how to extend the relation to product types and the fourth clause shows how to extend the relation to functional types. The last two conditions ensure that a logical relation relates fixed points. It is generally easy to prove that a relation is pointed: note that $\mathcal{R}^{T \times U}$ is pointed if both $\mathcal{R}^T$ and $\mathcal{R}^U$ are pointed and that $\mathcal{R}^{T \to U}$ is pointed if $\mathcal{R}^U$ is pointed. Similarly, $\mathcal{R}^{T \times U}$ and $\mathcal{R}^{T \to U}$ are chain-complete if both $\mathcal{R}^T$ and $\mathcal{R}^U$ are chain-complete.

Now, say, we are given two models of the simply typed λ-calculus. Then Lemma 2.20 below shows that the meaning of a term in one model is logically related to its meaning in the other model. This lemma is sometimes called the *Basic Lemma* of logical relations.

Lemma 2.20 Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two models of the simply typed λ-calculus. Let $\mathcal{R}$ be a logical relation over $\mathcal{E}_1$ and $\mathcal{E}_2$ and let η_1 and η_2 be environments for $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $(\eta_1(a), \eta_2(a)) \in \mathcal{R}^T$ for all $a \in \mathit{var}$ with $T = \text{type}\ a$, then

$$(\mathcal{E}_1[\![v :: V]\!]\eta_1, \mathcal{E}_2[\![v :: V]\!]\eta_2) \in \mathcal{R}^V$$

for every term v of type V.

Proof. We proceed by induction on the typing derivation of $v :: V$.

- **Case** $v = c :: T$: the statement holds since $\mathcal{R}$ respects constants.
- **Case** $v = a :: T$: it is easy to see that $(\mathcal{E}_1[\![a :: T]\!]\eta_1, \mathcal{E}_2[\![a :: T]\!]\eta_2) \in \mathcal{R}^T$ since we have assumed that $(\eta_1(a), \eta_2(a)) \in \mathcal{R}^T$.
- **Case** $v = (t_1, t_2) :: (T_1 \times T_2)$: by the induction hypothesis we have

  $$(\mathcal{E}_1[\![t_1 :: T_1]\!]\eta_1, \mathcal{E}_2[\![t_1 :: T_1]\!]\eta_2) \in \mathcal{R}^{T_1} \cap (\mathcal{E}_1[\![t_2 :: T_2]\!]\eta_1, \mathcal{E}_2[\![t_2 :: T_2]\!]\eta_2) \in \mathcal{R}^{T_2}.$$

  Now, since by definition $\mathcal{E}_1[\![(t_1, t_2) :: (T_1 \times T_2)]\!]\eta_1 = \pi_1$ such that $\mathbf{outl}_1\ \pi_1 = \mathcal{E}_1[\![t_1 :: T_1]\!]\eta_1$ and $\mathbf{outr}_1\ \pi_1 = \mathcal{E}_1[\![t_2 :: T_2]\!]\eta_1$ and similarly for $\mathcal{E}_2$ we have

  $$(\mathbf{outl}_1\ \pi_1, \mathbf{outl}_2\ \pi_2) \in \mathcal{R}^{T_1} \cap (\mathbf{outr}_1\ \pi_1, \mathbf{outr}_2\ \pi_2) \in \mathcal{R}^{T_2}$$

  and consequently $(\pi_1, \pi_2) \in \mathcal{R}^{T_1 \times T_2}$.
- **Case** $v = (\mathit{outl}\ t) :: T_1$: by the induction hypothesis we have

  $$(\mathcal{E}_1[\![t :: (T_1 \times T_2)]\!]\eta_1, \mathcal{E}_2[\![t :: (T_1 \times T_2)]\!]\eta_2) \in \mathcal{R}^{T_1 \times T_2}$$

  which immediately implies

  $$(\mathbf{outl}_1\ (\mathcal{E}_1[\![t :: (T_1 \times T_2)]\!]\eta_1), \mathbf{outl}_2\ (\mathcal{E}_2[\![t :: (T_1 \times T_2)]\!]\eta_2)) \in \mathcal{R}^{T_1}.$$

- **Case** $v = (\mathit{outr}\ t) :: T_2$: analogous.
- **Case** $v = (\lambda a \,.\, t) :: (S \to T)$: we have to show that

$$\begin{aligned}
&(\mathcal{E}_1[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta_1, \mathcal{E}_2[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta_2) \in \mathcal{R}^{S \to T} \\
&\quad \equiv \forall \delta_1\ \delta_2 \,.\, (\delta_1, \delta_2) \in \mathcal{R}^S \\
&\qquad \supset (\mathbf{app}_1\ (\mathcal{E}_1[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta_1)\ \delta_1, \mathbf{app}_2\ (\mathcal{E}_2[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta_2)\ \delta_2) \in \mathcal{R}^T
\end{aligned}$$

Assume that $(\delta_1, \delta_2) \in \mathcal{R}^S$. Since the modified environments $\eta_1(a := \delta_1)$ and $\eta_2(a := \delta_2)$ are related, we can invoke the induction hypothesis to obtain

$$(\mathcal{E}_1[\![t :: T]\!]\eta_1(a := \delta_1), \mathcal{E}_2[\![t :: T]\!]\eta_2(a := \delta_2)) \in \mathcal{R}^T$$

Now, since $\mathbf{app}_1\ (\mathcal{E}_1[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta_1)\ \delta_1 = \mathcal{E}_1[\![t :: T]\!]\eta_1(a := \delta_1)$ and similarly for $\mathcal{E}_2$, the proposition follows.

- **Case** $v = (t\ u) :: V$: by the induction hypothesis we have

$$\begin{aligned}
&(\mathcal{E}_1[\![t :: U \to V]\!]\eta_1, \mathcal{E}_2[\![t :: U \to V]\!]\eta_2) \in \mathcal{R}^{U \to V} \\
&\quad \equiv \forall \delta_1\ \delta_2 \,.\, (\delta_1, \delta_2) \in \mathcal{R}^U \\
&\qquad \supset (\mathbf{app}_1\ (\mathcal{E}_1[\![t :: U \to V]\!]\eta_1)\ \delta_1, \mathbf{app}_2\ (\mathcal{E}_2[\![t :: U \to V]\!]\eta_2)\ \delta_2) \in \mathcal{R}^V
\end{aligned}$$

and

$$(\mathcal{E}_1[\![u :: U]\!]\eta_1, \mathcal{E}_2[\![u :: U]\!]\eta_2) \in \mathcal{R}^V$$

which implies

$$(\mathbf{app}_1\ (\mathcal{E}_1[\![t :: U \to V]\!]\eta_1)\ (\mathcal{E}_1[\![u :: U]\!]\eta_1), \mathbf{app}_2\ (\mathcal{E}_2[\![t :: U \to V]\!]\eta_2)\ (\mathcal{E}_2[\![u :: U]\!]\eta_2)) \in \mathcal{R}^V.$$

- **Case** $v = (\mathit{fix}\ t) :: U$: by the induction hypothesis we have

$$\begin{aligned}
&(\mathcal{E}_1[\![t :: U \to U]\!]\eta_1, \mathcal{E}_2[\![t :: U \to U]\!]\eta_2) \in \mathcal{R}^{U \to U} \\
&\quad \equiv \forall \delta_1\ \delta_2 \,.\, (\delta_1, \delta_2) \in \mathcal{R}^U \\
&\qquad \supset (\mathbf{app}_1\ (\mathcal{E}_1[\![t :: U \to U]\!]\eta_1)\ \delta_1, \mathbf{app}_2\ (\mathcal{E}_2[\![t :: U \to U]\!]\eta_2)\ \delta_2) \in \mathcal{R}^U.
\end{aligned}$$

Since $\mathcal{R}^U$ is pointed, we have $(\bot, \bot) \in \mathcal{R}^U$. A straightforward induction shows that $(\delta^1_n, \delta^2_n) \in \mathcal{R}^U$ for all $n \in \mathbb{N}$. Since $\mathcal{R}^U$ is furthermore chain-complete, we have

$$(\bigsqcup\{\delta^1_n \mid n \in \mathbb{N}\}, \bigsqcup\{\delta^2_n \mid n \in \mathbb{N}\}) \in \mathcal{R}^U$$

as desired. □

An example application Let us conclude the section with an example application of logical relations. In fact, the purpose of the example is twofold. First, it illustrates the use of several notions we have introduced in this section. Second, it implies a useful result that we require in the following chapter.

For concreteness, let us assume that we have one type constant, say, *Nat* and two individual constants

$$\begin{aligned}
\mathit{zero} &:: \mathit{Nat} \\
\mathit{succ} &:: \mathit{Nat} \to \mathit{Nat}.
\end{aligned}$$

Furthermore, assume that we are given a type $P = P_1 \to \cdots \to P_n \to Nat$ and a model $\mathcal{E} = (\mathsf{E}, \mathsf{outl}, \mathsf{outr}, \mathsf{app}, \mathsf{const})$. Building upon $\mathcal{E}$ we will construct two other models, $\mathcal{K}$ and $\mathcal{L}$, and establish a relation between the two. To improve readability, we abbreviate $\mathsf{app}_{T,U}\ \varphi\ d$ by $\varphi \mathbin{\text{▪}} d$ and we omit $\mathsf{app}^{-1}_{T,U}$ altogether.

The first model, $\mathcal{K} = (\mathsf{K}, \mathsf{outl}^{\mathcal{K}}, \mathsf{outr}^{\mathcal{K}}, \mathsf{app}^{\mathcal{K}}, \mathsf{const}^{\mathcal{K}})$, is given by

$$
\begin{array}{lcl}
\mathsf{K}^T & = & \mathsf{E}^{P_1 \to \cdots \to P_n \to T} \\
\mathsf{outl}^{\mathcal{K}}\ \pi & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{outl}\ (\pi \mathbin{\text{▪}} \pi_1 \mathbin{\text{▪}} \cdots \mathbin{\text{▪}} \pi_n) \\
\mathsf{outr}^{\mathcal{K}}\ \pi & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{outr}\ (\pi \mathbin{\text{▪}} \pi_1 \mathbin{\text{▪}} \cdots \mathbin{\text{▪}} \pi_n) \\
\mathsf{app}^{\mathcal{K}}\ \varphi\ \delta & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, (\varphi \mathbin{\text{▪}} \pi_1 \mathbin{\text{▪}} \cdots \mathbin{\text{▪}} \pi_n) \mathbin{\text{▪}} (\delta \mathbin{\text{▪}} \pi_1 \mathbin{\text{▪}} \cdots \mathbin{\text{▪}} \pi_n) \\
\mathsf{const}^{\mathcal{K}}(c) & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{const}(c).
\end{array}
$$

Each element is interpreted by a function that takes n parameters; $\mathsf{app}^{\mathcal{K}}$ passes the parameters to both of its arguments while $\mathsf{const}^{\mathcal{K}}(c)$ ignores them. It is not hard to show that $\mathcal{K}$ is extensional (using the fact that $\mathcal{E}$ is extensional). How do we show that $\mathcal{K}$ satisfies the environment model condition? In fact, the easiest way to establish this condition is to use the combinatory model condition instead. Since $\mathcal{E}$ is a model it has combinators P, L, R, K and S. The combinators of $\mathcal{K}$ are simply given by

$$
\begin{array}{lcl}
\mathsf{P}^{\mathcal{K}} & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{P} \\
\mathsf{L}^{\mathcal{K}} & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{L} \\
\mathsf{R}^{\mathcal{K}} & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{R} \\
\mathsf{K}^{\mathcal{K}} & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{K} \\
\mathsf{S}^{\mathcal{K}} & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{S}.
\end{array}
$$

We leave it to the reader to check that the equational laws are satisfied.

To define the second model we require the following 'lifting' operation on types.

$$
\begin{array}{lcl}
\uparrow\! Nat & = & P \\
\uparrow\! T \times U & = & (\uparrow\! T) \times (\uparrow\! U) \\
\uparrow\! T \to U & = & (\uparrow\! T) \to (\uparrow\! U)
\end{array}
$$

The second model $\mathcal{L} = (\mathsf{L}, \mathsf{outl}^{\mathcal{L}}, \mathsf{outr}^{\mathcal{L}}, \mathsf{app}^{\mathcal{L}}, \mathsf{const}^{\mathcal{L}})$ is then given by

$$
\begin{array}{lcl}
\mathsf{L}^T & = & \mathsf{E}^{\uparrow T} \\
\mathsf{outl}^{\mathcal{L}}\ \pi & = & \mathsf{outl}\ \pi \\
\mathsf{outr}^{\mathcal{L}}\ \pi & = & \mathsf{outr}\ \pi \\
\mathsf{app}^{\mathcal{L}}\ \varphi\ \delta & = & \varphi \mathbin{\text{▪}} \delta \\
\mathsf{const}^{\mathcal{L}}(zero) & = & \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{const}(zero) \\
\mathsf{const}^{\mathcal{L}}(succ) & = & \boldsymbol{\lambda} \varphi \,.\, \boldsymbol{\lambda} \pi_1\ \ldots\ \pi_n \,.\, \mathsf{const}(succ) \mathbin{\text{▪}} (\varphi \mathbin{\text{▪}} \pi_1 \mathbin{\text{▪}} \cdots \mathbin{\text{▪}} \pi_n).
\end{array}
$$

Note that application in $\mathcal{L}$ is implemented by application in $\mathcal{E}$ albeit at a higher functional level: $\mathsf{app}^{\mathcal{L}}_{T,U} = \mathsf{app}_{\uparrow T, \uparrow U}$. Clearly, $\mathcal{L}$ is a model since $\mathcal{E}$ is one.

Now, note that $\mathsf{K}^{Nat} = \mathsf{L}^{Nat}$. In fact, $\mathcal{K}$ and $\mathcal{L}$ interpret a term of type Nat by the same element of E^P. More generally, $\mathcal{K}$ and $\mathcal{L}$ are related by the logical relation $(\sim^T) \subseteq \mathsf{K}^T \times \mathsf{L}^T$ given by

$$
\begin{array}{lcl}
\delta_1 \sim^{Nat} \delta_2 & \equiv & \delta_1 = \delta_2 \\
\pi_1 \sim^{T \times U} \pi_2 & \equiv & \mathsf{outl}^{\mathcal{K}}\ \pi_1 \sim^T \mathsf{outl}^{\mathcal{L}}\ \pi_2 \cap \mathsf{outr}^{\mathcal{K}}\ \pi_1 \sim^U \mathsf{outr}^{\mathcal{L}}\ \pi_2 \\
\varphi_1 \sim^{T \to U} \varphi_2 & \equiv & \forall \delta_1 \in \mathsf{K}^T, \delta_2 \in \mathsf{L}^T \,.\, \delta_1 \sim^T \delta_2 \supset \mathsf{app}^{\mathcal{K}}\ \varphi_1\ \delta_1 \sim^U \mathsf{app}^{\mathcal{L}}\ \varphi_2\ \delta_2.
\end{array}
$$

So $(\sim^T)$ is simply the extension of the equality relation on E^P.

THEOREM 2.21 Let $t :: T$ be a closed term, then

$$\mathcal{K}[\![t]\!] \sim^T \mathcal{L}[\![t]\!].$$

PROOF. It is not hard to see that $(\sim^T)$ is both pointed and chain-complete. It remains to prove that $(\sim^T)$ relates constants: we have to show that

$$\begin{array}{lcl} \mathsf{const}^{\mathcal{K}}(zero) & = & \mathsf{const}^{\mathcal{L}}(zero) \\ \mathsf{app}^{\mathcal{K}}\ (\mathsf{const}^{\mathcal{K}}(succ))\ \delta & = & \mathsf{app}^{\mathcal{L}}\ (\mathsf{const}^{\mathcal{L}}(succ))\ \delta. \end{array}$$

The first equation obviously holds and the latter equation follows directly from $\mathsf{const}^{\mathcal{L}}(succ) = \boldsymbol{\lambda}\varphi\,.\,\mathsf{app}^{\mathcal{K}}\ (\mathsf{const}^{\mathcal{K}}(succ))\ \varphi$. □

The effect of the two interpretations $\mathcal{K}$ and $\mathcal{L}$ can also be expressed on a syntactical level. Define

$$\begin{array}{lcl} \hat{K}\ t & = & \lambda x_1\ \ldots\ x_n\,.\,t \\ \hat{S}\ t\ u & = & \lambda x_1\ \ldots\ x_n\,.\,(t\ x_1\ \ldots\ x_n)\ (u\ x_1\ \ldots\ x_n) \end{array}$$

then we have $\mathcal{K}[\![t]\!] = \mathcal{E}[\![\hat{K}\ t]\!]$ for all closed terms t. The proof proceeds by induction over the structure of so-called *combinatory terms* (that is, terms built from $P = \lambda x\ y\,.\,(x, y)$, $L = \lambda z\,.\,outl\ z$, $R = \lambda z\,.\,outr\ z$, $K = \lambda x\ y\,.\,x$, $S = \lambda x\ y\ z\,.\,(x\ z)\ (y\ z)$ and constants using application) employing the fact that $\hat{S}\ (\hat{K}\ t)\ (\hat{K}\ u) = \hat{K}\ (t\ u)$.

The second model corresponds to a program transformation called *lifting*. Lifting maps a term $t :: T$ to a term $\uparrow t :: \uparrow T$ where $\uparrow t$ is defined as follows (we assume that for each variable a of type T there is a lifted variable named $\underline{a}$ of type $\uparrow T$):

$$\begin{array}{lcl} \uparrow c & = & \underline{c} \\ \uparrow a & = & \underline{a} \\ \uparrow(t_1, t_2) & = & (\uparrow t_1, \uparrow t_2) \\ \uparrow outl\ t & = & outl\ (\uparrow t) \\ \uparrow outr\ t & = & outr\ (\uparrow t) \\ \uparrow \lambda a\,.\,t & = & \lambda \underline{a}\,.\uparrow t \\ \uparrow t\ u & = & (\uparrow t)\ (\uparrow u) \\ \uparrow fix\ t & = & fix\ (\uparrow t). \end{array}$$

The lifted versions of the constants $zero$ and $succ$ are given by

$$\begin{array}{lcl} \underline{zero} & = & \lambda x_1\ \ldots\ x_n\,.\,zero \\ \underline{succ}\ n & = & \lambda x_1\ \ldots\ x_n\,.\,succ\ (n\ x_1\ \ldots\ x_n). \end{array}$$

It is not hard to show that $\mathcal{L}[\![t]\!] = \mathcal{E}[\![\uparrow t]\!]$ for all closed terms t (in general, we have $\mathcal{L}[\![t]\!]\eta = \mathcal{E}[\![\uparrow t]\!]\underline{\eta}$ where $\eta(a) = \underline{\eta}(\underline{a})$). Now putting everything together we obtain the following corollary of Theorem 2.21.

COROLLARY 2.22 Let $t :: T$ be a closed term, then

$$\mathcal{E}[\![\hat{K}\ t]\!] \sim^T \mathcal{E}[\![\uparrow t]\!].$$ □

2.5 The polymorphic λ-calculus

Considered as a programming language the simply typed λ-calculus is very restrictive. For instance, while we can form a pair of values of arbitrary types, we cannot define a single function that swaps elements of an arbitrary pair. The typing rules require that we precisely lay down the types of the components. The *swap* function cries for *polymorphism*. In fact, polymorphism nicely complements the type security of the simply typed λ-calculus with flexibility. A polymorphic type system like the one introduced in this section allows the definition of functions like *swap* that behave uniformly over all types.

The polymorphic λ-calculus builds upon the simply typed λ-calculus in two ways. On the value level it extends the simply typed λ-calculus by constructions for creating and using polymorphic values. On the type level it reuses the simply typed λ-calculus: the type terms of the polymorphic λ-calculus are essentially the terms of the simply typed λ-calculus.

The polymorphic λ-calculus has been discovered independently by Girard (1972) and Reynolds (1974). It trades under a variety of names: second-order λ-calculus or system $F2$ (in these cases A in $\forall A \,.\, T$ is restricted to types of kind $\star$), higher-order λ-calculus or system $F\omega$. Apart from its use as a model for polymorphism the polymorphic λ-calculus is also used in practice as the internal language of the Glasgow Haskell Compiler (Peyton Jones 1996).

2.5.1 Syntax

Syntactic categories The polymorphic λ-calculus has a three-level structure.

kind terms	$\mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind}$
type constants	$C, D \in \mathit{Const}$
type variables	$A, B \in \mathit{Var}$
type terms	$T, U \in \mathit{Type}$
individual constants	$c, d \in \mathit{const}$
individual variables	$a, b \in \mathit{var}$
terms	$t, u \in \mathit{term}$

We use upper-case Fraktur letters for kinds, upper-case Roman letters for types and lower-case Roman letters for terms.

Kinds Kind terms are formed according to the following grammar.

$$\begin{array}{rcll} \mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind} & ::= & \star & \text{kind of types} \\ & | & \mathfrak{T} \times \mathfrak{U} & \text{product kind} \\ & | & \mathfrak{T} \to \mathfrak{U} & \text{function kind} \end{array}$$

Thus, the kind terms of the polymorphic λ-calculus are the type terms of the simply typed λ-calculus. The kind '$\star$' represents the 'type' of manifest types such as *Char* or *Int*. The *kind formation rules* are displayed in Figure 2.4. Here, '$\Box$' denotes the 'type' of kinds, sometimes called *superkind*.

Types Type terms are built from type constants and type variables using type pairing, type projection, type application, type abstraction, type recursion and construction of polymorphic types. As before, we assume that type constants and type variables are kinded, that is, they are annotated with their kinds, usually written $S :: \mathfrak{T}$. If $S :: \mathfrak{T}$ is a kinded type constant or type variable, we define 'kind $(S :: \mathfrak{T}) = \mathfrak{T}$'. *Pseudo-type terms* are formed according to the following

$$\frac{}{\star :: \square}\ (\text{K-}\star\text{-FORM})$$

$$\frac{\mathfrak{T} :: \square \qquad \mathfrak{U} :: \square}{\mathfrak{T} \times \mathfrak{U} :: \square}\ (\text{K-}\times\text{-FORM}) \qquad \frac{\mathfrak{T} :: \square \qquad \mathfrak{U} :: \square}{\mathfrak{T} \to \mathfrak{U} :: \square}\ (\text{K-}\to\text{-FORM})$$

Figure 2.4: Kind formation rules.

grammar.

$$\begin{array}{rcll} T, U \in \mathit{Type} & ::= & C & \text{type constant} \\ & | & A & \text{type variable} \\ & | & (T_1, T_2) & \text{type pairing} \\ & | & \mathit{Outl}\ T & \text{type projection} \\ & | & \mathit{Outr}\ T & \text{type projection} \\ & | & \Lambda A\,.\,T & \text{type abstraction} \\ & | & T\ U & \text{type application} \\ & | & \mathit{Fix}\ T & \text{type recursion} \\ & | & \forall A\,.\,T & \text{polymorphic type} \end{array}$$

Thus, the pseudo-type terms of the polymorphic λ-calculus are essentially the pseudo-terms of the simply typed λ-calculus. The only addition is a construction for polymorphic types, which gives the polymorphic λ-calculus its name.

The choice of *Const*, the set of type constants, is more or less arbitrary. Of course, *Const* should contain at least the function space constructor. For concreteness, we assume that *Const* comprises the following type constants ('1', '+', '×' are included so that we can model Haskell data type declarations, see below):

$$\begin{array}{lcl} \mathit{Char} & :: & \star \\ \mathit{Int} & :: & \star \\ 1 & :: & \star \\ (+) & :: & \star \to \star \to \star \\ (\times) & :: & \star \to \star \to \star \\ (\to) & :: & \star \to \star \to \star. \end{array}$$

We assume that '→', '×' and '+' associate to the right. Furthermore, '→' binds more tightly than '×', which takes precedence over '+'.

A pseudo-type term T is called a *type term* if there is some kind $\mathfrak{T}$ such that $T :: \mathfrak{T}$ is derivable using the *kinding rules* depicted in Figure 2.5. Note that A in $\forall A\,.\,T$ may range over any kind. A type-term is called *monomorphic* if it does not contain any occurrences of '∀'. The set of all monomorphic type-terms is denoted *MonoType* (for emphasis the set of all type terms is sometimes denoted *PolyType*). Define $\star^n \to \star$ by $\star^0 \to \star = \star$ and $\star^{n+1} \to \star = \star \to (\star^n \to \star)$. If T has kind $\star^n \to \star$, we say that T has *arity* n. The *rank* (McCracken 1984) of a type term is given by (the other cases are the obvious ones):

$$\begin{array}{lcl} \mathit{rank}(A) & = & 0 \\ \mathit{rank}(\forall A\,.\,T) & = & \mathit{max}\{1, \mathit{rank}(T)\} \\ \mathit{rank}(T \to U) & = & \mathit{max}\{1 + \mathit{rank}(T), \mathit{rank}(U)\}. \end{array}$$

Finally, we transfer the relation '≈' (see Definition 2.18) to type terms (additionally setting $T \approx U \supset \forall A\,.\,T \approx \forall A\,.\,U$).

$$\frac{}{C :: \text{kind } C}\ (\textsc{t-const}) \qquad \frac{}{A :: \text{kind } A}\ (\textsc{t-var})$$

$$\frac{T_1 :: \mathfrak{T}_1 \qquad T_2 :: \mathfrak{T}_2}{(T_1, T_2) :: (\mathfrak{T}_1 \times \mathfrak{T}_2)}\ (\textsc{t-}\times\textsc{-intro})$$

$$\frac{T :: (\mathfrak{T}_1 \times \mathfrak{T}_2)}{(\mathit{Outl}\ T) :: \mathfrak{T}_1}\ (\textsc{t-}\times\textsc{-elim-l}) \qquad \frac{T :: (\mathfrak{T}_1 \times \mathfrak{T}_2)}{(\mathit{Outr}\ T) :: \mathfrak{T}_2}\ (\textsc{t-}\times\textsc{-elim-r})$$

$$\frac{T :: \mathfrak{T}}{(\Lambda A \,.\, T) :: (\text{kind } A \to \mathfrak{T})}\ (\textsc{t-}\to\textsc{-intro})$$

$$\frac{T :: (\mathfrak{U} \to \mathfrak{V}) \qquad U :: \mathfrak{U}}{(T\ U) :: \mathfrak{V}}\ (\textsc{t-}\to\textsc{-elim})$$

$$\frac{T :: \mathfrak{U} \to \mathfrak{U}}{(\mathit{Fix}\ T) :: \mathfrak{U}}\ (\textsc{t-rec}) \qquad \frac{T :: \star}{(\forall A \,.\, T) :: \star}\ (\textsc{t-all})$$

Figure 2.5: Kinding rules.

Here are some type terms that will be used in the subsequent chapters:

$$\begin{array}{lcl}
\mathit{Id} & :: & \star \to \star \\
\mathit{Id} & = & \Lambda A :: \star \,.\, A \\[4pt]
K & :: & \star \to \star \to \star \\
K & = & \Lambda A :: \star \,.\, \Lambda B :: \star \,.\, A \\[4pt]
(\cdot) & :: & (\star \to \star) \to (\star \to \star) \to (\star \to \star) \\
F \cdot G & = & \Lambda A :: \star \,.\, F\ (G\ A) \\[4pt]
\underline{1}, \underline{\mathit{Char}}, \underline{\mathit{Int}} & :: & \star \to \star \\
\underline{1} & = & \Lambda A :: \star \,.\, 1 \\
\underline{\mathit{Char}} & = & \Lambda A :: \star \,.\, \mathit{Char} \\
\underline{\mathit{Int}} & = & \Lambda A :: \star \,.\, \mathit{Int} \\[4pt]
(\underline{+}), (\underline{\times}), (\underline{\to}) & :: & (\star \to \star) \to (\star \to \star) \to (\star \to \star) \\
F \mathbin{\underline{+}} G & = & \Lambda A :: \star \,.\, F\ A + G\ A \\
F \mathbin{\underline{\times}} G & = & \Lambda A :: \star \,.\, F\ A \times G\ A \\
F \mathbin{\underline{\to}} G & = & \Lambda A :: \star \,.\, F\ A \to G\ A.
\end{array}$$

Note that we take some notational liberties: we write $F\ A = T$ instead of $F = \Lambda A \,.\, T$ and we often omit kind annotations of type constants and type variables (usually the kind of a type variable is only given in the binding position).

The type language is fairly expressive. It subsumes, for instance, the type system of Haskell. As an example, we can easily translate Haskell data type declarations into type terms. Recall the schematic form of **data** declarations given in Section 2.1:

$$\textbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}.$$

The type B thus defined can be written as the following type term (we tacitly assume that the kinds of the type variables have been inferred)

$$\mathit{Fix}\ (\Lambda B \,.\, \Lambda A_1\ \ldots\ A_m \,.\, (T_{11} \times \cdots \times T_{1m_1}) + \cdots + (T_{n1} \times \cdots \times T_{nm_n})),$$

where $T_1 \times \cdots \times T_k = 1$ for $k = 0$. For simplicity, n-ary sums are reduced to binary sums and n-ary products to binary products. For instance, the **data** declarations

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{List}\ A & = & \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A) \\
\mathbf{data}\ \mathit{Fork}\ A & = & \mathit{fork}\ A\ A \\
\mathbf{data}\ \mathit{Perfect}\ A & = & \mathit{zeroP}\ A \mid \mathit{succP}\ (\mathit{Perfect}\ (\mathit{Fork}\ A))
\end{array}$$

are translated to (see also Example 2.17)

$$\begin{array}{lcl}
\mathit{List} & :: & \star \to \star \\
\mathit{List} & = & \mathit{Fix}\ (\Lambda \mathit{List} \,.\, \Lambda A \,.\, 1 + A \times \mathit{List}\ A) \\
\mathit{Fork} & :: & \star \to \star \\
\mathit{Fork} & = & \Lambda A \,.\, A \times A \\
\mathit{Perfect} & :: & \star \to \star \\
\mathit{Perfect} & = & \mathit{Fix}\ (\Lambda \mathit{Perfect} \,.\, \Lambda A \,.\, A + \mathit{Perfect}\ (\mathit{Fork}\ A)).
\end{array}$$

Note that we have simplified $\mathit{Fix}\ (\Lambda \mathit{Fork} \,.\, \Lambda A \,.\, A \times A)$ to $\Lambda A \,.\, A \times A$.

Interestingly, the representation of regular types such as List can be improved by applying a technique called *lambda-dropping* (Danvy 1999): if $\mathit{Fix}\ (\Lambda F \,.\, \Lambda A \,.\, T)$ is regular, then it is equivalent to $\Lambda A \,.\, \mathit{Fix}\ (\Lambda B \,.\, T[F\ \ A := B])$ where $T[\,T_1 := T_2\,]$ denotes the type term, in which all occurrences of T_1 are replaced by T_2. For instance, the λ-dropped version of $\mathit{Fix}\ (\Lambda \mathit{List} \,.\, \Lambda A \,.\, 1 + A \times \mathit{List}\ A)$ is $\Lambda A \,.\, \mathit{Fix}\ (\Lambda B \,.\, 1 + A \times B)$. The λ-dropped version employs the fixed point operator at kind $\star$ (that is, the subterm $\mathit{Fix}\ T$ has kind $\star$) whereas the λ-lifted version employs the fixed point operator at kind $\star \to \star$. Nested types such as $\mathit{Perfect}$ are not amenable to this transformation since the type argument of the nested type is changed in the recursive call(s). As an aside, note that the λ-dropped and the λ-lifted version correspond to two different methods of modelling parameterized types: families of first-order fixed points versus higher-order fixed points, see, for instance, (Bird and Paterson 1999).

We have not yet taken into account that data type definitions can be mutually recursive. Fortunately, since the language of types provides pairs (where pair means pair of types, *not* product type), we can easily deal with the general case. Say, we are given two recursive equations $B_1 = T_1$ and $B_2 = T_2$, then we can express B_1 and B_2 using fixed points operating on pairs:

$$\begin{array}{lcl}
B_1 & = & \mathit{Outl}\ (\mathit{Fix}\ (\Lambda B \,.\, (T_1[B_1 := \mathit{Outl}\ B, B_2 := \mathit{Outr}\ B], T_2[B_1 := \mathit{Outl}\ B, B_2 := \mathit{Outr}\ B]))) \\
B_2 & = & \mathit{Outr}\ (\mathit{Fix}\ (\Lambda B \,.\, (T_1[B_1 := \mathit{Outl}\ B, B_2 := \mathit{Outr}\ B], T_2[B_1 := \mathit{Outl}\ B, B_2 := \mathit{Outr}\ B]))).
\end{array}$$

Likewise a system of n recursive equations can be dealt with using n-tuples (or nested pairs).

REMARK 2.23 An alternative approach taken in (Hinze 2000f) is to introduce recursion equations into the type language.

$$\begin{array}{lcll}
T, U \in \mathit{Type} & ::= & \ldots & \\
 & | & T\ \mathbf{where}\ \{A_1 = T_1; \ldots; A_n = T_n\} & \text{local type definition}
\end{array}$$

While this approach allows us to model data type declarations more directly and also more naturally, it complicates the development in later chapters. □

Terms As before, we assume that constants and variables are annotated with their types. Of course, the type of a constant must be closed. *Pseudo-terms* are formed according to the following grammar.

$$
\begin{array}{rcll}
t, u \in \mathit{term} & ::= & c & \text{constant} \\
& | & a & \text{variable} \\
& | & () & \text{empty tuple} \\
& | & \mathit{inl}\ t_1 & \text{injection} \\
& | & \mathit{inr}\ t_2 & \text{injection} \\
& | & \multicolumn{2}{l}{\textbf{case}\ t\ \textbf{of}\ \{\mathit{inl}\ a_1 \Rightarrow u_1 ; \mathit{inr}\ a_2 \Rightarrow u_2\}} \\
& & & \lfloor\text{case analysis} \\
& | & (t_1, t_2) & \text{pairing} \\
& | & \mathit{outl}\ t & \text{projection} \\
& | & \mathit{outr}\ t & \text{projection} \\
& | & \lambda a \,.\, t & \text{abstraction} \\
& | & t\ u & \text{application} \\
& | & \lambda A \,.\, t & \text{universal abstraction} \\
& | & t\ U & \text{universal application} \\
& | & \mathit{fix}\ t & \text{recursion}
\end{array}
$$

Here, $\lambda A \,.\, t$ denotes universal abstraction (forming a polymorphic value) and $t\ U$ denotes universal application (instantiating a polymorphic value). Note that we use the same syntax for value abstraction $\lambda a \,.\, t$ (here a is a value variable) and universal abstraction $\lambda A \,.\, t$ (here A is a type variable). The term language contains constructs for the type constants '1', '+', '×' and '→'. We assume that the set *const* of value constants includes suitable functions for each of the other type constants C in *Const*.

Remark 2.24 The syntax of the polymorphic λ-calculus is slightly different from Haskell syntax: the abstraction $\lambda a \,.\, t$ is written $\lambda a \to t$ in Haskell and the case analysis **case** t **of** $\{\mathit{inl}\ a_1 \Rightarrow u_1 ; \mathit{inr}\ a_2 \Rightarrow u_2\}$ is written **case** t **of** $\{\mathit{inl}\ a_1 \to u_1 ; \mathit{inr}\ a_2 \to u_2\}$—in general, we avoid using the arrow '→' too often. □

A pseudo-term t is called a *term* if there is some type T such that $t :: T$ is derivable using the *typing rules* depicted in Figure 2.6. Two remarks are in order. First, the restriction on type variables in rule (∀-intro) prevents non-sensible terms such as $\lambda A :: \star \,.\, a :: A$ where the value variable a carries the type variable A out of scope.

Second, rule (conv) allows to interchange types which are structurally equivalent, that is, which have the same Böhm tree (see Definition 2.18). Note that this is a very liberal notion of type equivalence. Consider, for instance, List_1 and List_2 given by (the λ-lifted and λ-dropped versions of *List*)

$$
\begin{array}{rcl}
\mathit{List}_1 & = & \mathit{Fix}\ (\Lambda \mathit{List} \,.\, \Lambda A \,.\, 1 + A \times \mathit{List}\ A) \\
\mathit{List}_2 & = & \Lambda A \,.\, \mathit{Fix}\ (\Lambda B \,.\, 1 + A \times B).
\end{array}
$$

We have $\mathit{List}_1\ \mathit{Char} \approx \mathit{List}_2\ \mathit{Char}$, but $\mathit{List}_1\ \mathit{Char}$ and $\mathit{List}_2\ \mathit{Char}$ are, for instance, not convertible. Furthermore, note since the relation $(\approx)$ is undecidable in general, we have an undecidable type system.

Remark 2.25 The term language is quite voluminous. A less involved alternative is to introduce the constructs for the type constants '1', '+' and '×' as additional

$$\frac{}{c :: \mathrm{type}\ c}\ (\textsc{var}) \qquad \frac{}{a :: \mathrm{type}\ a}\ (\textsc{const})$$

$$\frac{}{() :: 1}\ (\textsc{1-intro})$$

$$\frac{t_1 :: T_1}{(\mathit{inl}\ t_1) :: (T_1 + T_2)}\ (\textsc{+-intro-l}) \qquad \frac{t_2 :: T_2}{(\mathit{inr}\ t_2) :: (T_1 + T_2)}\ (\textsc{+-intro-r})$$

$$\frac{t :: (\mathrm{type}\ a_1 + \mathrm{type}\ a_2) \qquad u_1 :: U \qquad u_2 :: U}{(\textbf{case}\ t\ \textbf{of}\ \{\mathit{inl}\ a_1 \Rightarrow u_1; \mathit{inr}\ a_2 \Rightarrow u_2\}) :: U}\ (\textsc{+-elim})$$

$$\frac{t_1 :: T_1 \qquad t_2 :: T_2}{(t_1, t_2) :: (T_1 \times T_2)}\ (\textsc{×-intro})$$

$$\frac{t :: (T_1 \times T_2)}{(\mathit{outl}\ t) :: T_1}\ (\textsc{×-elim-l}) \qquad \frac{t :: (T_1 \times T_2)}{(\mathit{outr}\ t) :: T_2}\ (\textsc{×-elim-r})$$

$$\frac{t :: T}{(\lambda a\,.\,t) :: (\mathrm{type}\ a \to T)}\ (\textsc{→-intro}) \qquad \frac{t :: (U \to V) \qquad u :: U}{(t\ u) :: V}\ (\textsc{→-elim})$$

$$\frac{t :: T}{(\lambda A\,.\,t) :: (\forall A\,.\,T)}\ \begin{array}{l} A \text{ not free in the type} \\ \text{of a free variable of } t \end{array}\ (\textsc{∀-intro})$$

$$\frac{t :: (\forall A\,.\,V) \qquad U :: \mathrm{kind}\ A}{(t\ U) :: V[A := U]}\ (\textsc{∀-elim})$$

$$\frac{t :: U \to U}{(\mathit{fix}\ t) :: U}\ (\textsc{fix})$$

$$\frac{t :: T \qquad T \approx U}{t :: U}\ (\textsc{conv})$$

Figure 2.6: Typing rules.

constants:

$$\begin{array}{lcl}
() & :: & 1 \\
\mathit{inl} & :: & \forall A_1\ A_2 \,.\, A_1 \to A_1 + A_2 \\
\mathit{inr} & :: & \forall A_1\ A_2 \,.\, A_2 \to A_1 + A_2 \\
\mathbf{case} & :: & \forall A_1\ A_2\ B \,.\, A_1 + A_2 \to (A_1 \to B) \to (A_2 \to B) \to B \\
(,) & :: & \forall A_1\ A_2 \,.\, A_1 \to A_2 \to A_1 \times A_2 \\
\mathit{outl} & :: & \forall A_1\ A_2 \,.\, A_1 \times A_2 \to A_1 \\
\mathit{outr} & :: & \forall A_1\ A_2 \,.\, A_1 \times A_2 \to A_2.
\end{array}$$

A drawback of this approach is that *inl*, *inr* etc now take two additional type arguments. We will use the variant whichever is more appropriate. □

Let us finally look at some examples:

$$\begin{array}{lcl}
\mathit{id} & :: & \forall A :: \star \,.\, A \to A \\
\mathit{id} & = & \lambda A :: \star \,.\, \lambda a :: A \,.\, a \\
k & :: & \forall A :: \star \,.\, \forall B :: \star \,.\, A \to B \to A \\
k & = & \lambda A :: \star \,.\, \lambda B :: \star \,.\, \lambda a :: A \,.\, \lambda b :: B \,.\, a \\
(\triangledown) & :: & \forall A\ B\ C \,.\, (A \to C) \to (B \to C) \to (A + B \to C) \\
f \triangledown g & = & \lambda x \,.\, \mathbf{case}\ x\ \mathbf{of}\ \{\mathit{inl}\ a \Rightarrow f\ a; \mathit{inr}\ b \Rightarrow g\ b\} \\
\mathit{mapList} & :: & \forall A_1 :: \star \,.\, \forall A_2 :: \star \,.\, (A_1 \to A_2) \to (\mathit{List}\ A_1 \to \mathit{List}\ A_2) \\
\mathit{mapList} & = & \lambda A_1 :: \star \,.\, \lambda A_2 :: \star \,.\, \lambda \mathit{mapA} :: (A_1 \to A_2) \,.\, \lambda \mathit{as} :: \mathit{List}\ A_1 \,. \\
& & \mathbf{case}\ \mathit{as}\ \mathbf{of}\ \{\mathit{inl}\ u \Rightarrow \mathit{inl}\ u; \\
& & \qquad\qquad \mathit{inr}\ z \Rightarrow \mathit{inr}\ (\mathit{mapA}\ (\mathit{outl}\ z), \mathit{mapList}\ A_1\ A_2\ \mathit{mapA}\ (\mathit{outr}\ z))\}.
\end{array}$$

As usual, we take some notational liberties: we write $f\ a = t$ for $f = \lambda a \,.\, t$, we omit kind and type annotations and we sometimes omit universal abstractions and applications—especially when defining operators such as $(\triangledown)$.

2.5.2 Semantics

This section sketches the denotational semantics of the polymorphic λ-calculus. As in Section 2.4.2 we will use the general framework of environment models for the presentation. The semantics will be given in two steps. First, we define the semantics of types. Since type terms of the polymorphic λ-calculus are essentially terms of the simply typed λ-calculus, we will, in fact, re-use the semantics given in Section 2.4.2. Second, we define the semantics of terms.

Since we allow recursion both on the term and on the type level, we require domain-theoretic models both for terms and for types. Note that finding models that support solving arbitrary domain equations is by no means trivial. Suitable models are, for instance, models based on universal domains. These models interpret types as certain elements (closures, finitary retracts or finitary projections) of the universal domain, so that type recursion can be interpreted by the untyped least fixed point operator.

A particular attractive model is the *finitary projection model* (Amadio, Bruce, and Longo 1986) where types are represented by finitary projections. Briefly, a *projection* π is a continuous function that is *idempotent*, $\pi \cdot \pi = \mathit{id}$, and *reductive*, $\pi \sqsubseteq \mathit{id}$. A projection is *finitary* if its range is a domain. The central idea of this model is to interpret the type constraint $t :: T$ by the application $\pi\ [\![t]\!]$, where the finitary projection $\pi = [\![T]\!]$ coerces $[\![t]\!]$ to an element of the domain associated with T, that is, π's range. Now, if $[\![t]\!]$ is already an element of this domain, then π will leave it unchanged (since π is idempotent).

Types For simplicity, we use frames rather than applicative structures for the semantics of types.

DEFINITION 2.26 A kind frame $\mathcal{T}$ is a tuple $(\mathbf{T}, \mathbf{Const}, \Pi)$ such that

- $\mathbf{T} = (\mathbf{T}^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ is a family of domains, such that $\mathbf{T}^{\mathfrak{T} \times \mathfrak{U}} \subseteq \mathbf{T}^{\mathfrak{T}} \times \mathbf{T}^{\mathfrak{U}}$ and $\mathbf{T}^{\mathfrak{T} \to \mathfrak{U}} \subseteq \mathbf{T}^{\mathfrak{T}} \to \mathbf{T}^{\mathfrak{U}}$,
- $\mathbf{Const} : \mathit{Const} \to \mathbf{T}$ is a mapping from type constants to values such that $\mathbf{Const}(C) \in \mathbf{T}^{\mathfrak{T}}$ for all $C \in \mathit{Const}$ with $\mathfrak{T} = \text{kind } C$,
- $\Pi = (\Pi^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ is a family of continuous functions $\Pi^{\mathfrak{T}} \in \mathbf{T}^{(\mathfrak{T} \to \star) \to \star}$. $\square$

The elements of $\mathbf{T}^{\mathfrak{T}}$ represent type constructors. In particular, the elements of $\mathbf{T}^{\star}$ represent types (but note: they are not types, they merely *represent* types). For instance, in the finitary projection model the elements of $\mathbf{T}^{\mathfrak{T}}$ are finitary projections. The function $\Pi^{\mathfrak{T}}$ will be used to give a semantics to polymorphic types of the form $\forall A \,.\, T$ where A ranges over type constructors of kind $\mathfrak{T}$.

DEFINITION 2.27 A kind frame $\mathcal{T} = (\mathbf{T}, \mathbf{Const}, \Pi)$ is a *type model* if the clauses below define a total meaning function for types, where the meaning function is defined by induction on the structure of kinding derivations.

$$
\begin{array}{lcl}
\mathcal{T}[\![T :: \mathfrak{T}]\!]\eta & \in & \mathbf{T}^{\mathfrak{T}} \\
\mathcal{T}[\![C :: \mathfrak{C}]\!]\eta & = & \mathbf{Const}(C) \\
\mathcal{T}[\![A :: \mathfrak{A}]\!]\eta & = & \eta(A) \\
\mathcal{T}[\![(T_1, T_2) :: (\mathfrak{T}_1 \times \mathfrak{T}_2)]\!]\eta & = & (\mathcal{T}[\![T_1 :: \mathfrak{T}_1]\!]\eta, \mathcal{T}[\![T_2 :: \mathfrak{T}_2]\!]\eta) \\
\mathcal{T}[\![(\mathit{Outl}\ T) :: \mathfrak{T}_1]\!]\eta & = & \mathbf{outl}\ (\mathcal{T}[\![T :: \mathfrak{T}_1 \times \mathfrak{T}_2]\!]\eta) \\
\mathcal{T}[\![(\mathit{Outr}\ T) :: \mathfrak{T}_2]\!]\eta & = & \mathbf{outr}\ (\mathcal{T}[\![T :: \mathfrak{T}_1 \times \mathfrak{T}_2]\!]\eta) \\
\mathcal{T}[\![(\Lambda A \,.\, T) :: (\mathfrak{V} \to \mathfrak{T})]\!]\eta & = & \boldsymbol{\lambda}\alpha \in \mathbf{T}^{\mathfrak{V}} \,.\, \mathcal{T}[\![T :: \mathfrak{T}]\!]\eta(A := \alpha) \\
\mathcal{T}[\![(T\ U) :: \mathfrak{V}]\!]\eta & = & (\mathcal{T}[\![T :: \mathfrak{U} \to \mathfrak{V}]\!]\eta)\ (\mathcal{T}[\![U :: \mathfrak{U}]\!]\eta) \\
\mathcal{T}[\![(\mathit{Fix}\ T) :: \mathfrak{U}]\!]\eta & = & \mathbf{lfp}\ (\mathcal{T}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\eta) \\
\mathcal{T}[\![(\forall A \,.\, T) :: \star]\!]\eta & = & \Pi^{\mathfrak{T}}\ (\boldsymbol{\lambda}\alpha \in \mathbf{T}^{\mathfrak{T}} \,.\, \mathcal{T}[\![T :: \star]\!]\eta(A := \alpha)) \\
 & & \text{where } \mathfrak{T} = \text{kind } A.
\end{array}
$$

$\square$

Here, $\mathbf{lfp}$ is the least fixed point operator given by

$$
\begin{array}{lcl}
\mathbf{lfp}\ \varphi & = & \bigsqcup\{\alpha_n \mid n \in \mathbb{N}\} \\
\text{where } \alpha_0 & = & \bot \\
\alpha_{n+1} & = & \varphi\ \alpha_n .
\end{array}
$$

In the next chapter we require an extension of the meaning function that also interprets 'infinite type terms', that is, Böhm-like trees over the language of types. From the theory of infinite trees (Courcelle 1983) we know that every function that maps finite trees to elements of some domain can be uniquely extended to a continuous function on infinite trees. The following fact adapts the result to the current setting.

FACT 2.28 The meaning function for types can be uniquely extended to a continuous function on Böhm-like trees such that $\mathcal{T}[\![\Omega]\!]\eta = \bot$ and

$$\mathcal{T}[\![\mathrm{BT}(T)]\!]\eta = \mathcal{T}[\![T]\!]\eta$$

for all types $T \in \mathit{Type}$ and all environments η. $\square$

A simple consequence of this fact is that structurally equivalent types are interpreted by the same element of $\mathbf{T}$.

Corollary 2.29 Let T_1 and T_2 be two type terms of kind $\mathfrak{T}$, then

$$T_1 \approx T_2 \quad \supset \quad \forall \eta \,.\, \mathcal{T}[\![T_1]\!]\eta = \mathcal{T}[\![T_2]\!]\eta.$$

Proof. We assume that $T_1 \approx T_2$ and reason:

$$\begin{array}{ll} & \mathcal{T}[\![T_1]\!]\eta \\ = & \quad \{ \text{ Fact 2.28 } \} \\ & \mathcal{T}[\![\mathrm{BT}(T_1)]\!]\eta \\ = & \quad \{ \; T_1 \approx T_2 \; \} \\ & \mathcal{T}[\![\mathrm{BT}(T_2)]\!]\eta \\ = & \quad \{ \text{ Fact 2.28 } \} \\ & \mathcal{T}[\![T_2]\!]\eta. \end{array}$$

□

Terms The semantics of terms will be specified only for a fragment of the language. In particular, we do not consider any constructs associated with the type constants '1', '+' and '×'. We tacitly assume that these constructs are supplied as additional constants, see Remark 2.25.

Definition 2.30 An *applicative structure for the polymorphic λ-calculus* $\mathcal{E}$ is a tuple $(\mathcal{T}, \mathit{Dom}, \mathbf{app}, \mathbf{uapp}, \mathbf{const})$ such that

- $\mathcal{T} = (\mathbf{T}, \mathbf{Const}, \Pi)$ is a kind frame,
- $\mathit{Dom} = (\mathit{Dom}^{\alpha} \mid \alpha \in \mathbf{T}^{\star})$ is a family of domains,
- $\mathbf{app} = (\mathbf{app}_{\alpha,\beta} \mid \alpha, \beta \in \mathbf{T}^{\star})$ is a family of continuous functions with $\mathbf{app}_{\alpha,\beta} : [\mathit{Dom}^{\mathbf{Const}(\to)\ \alpha\ \beta} \to [\mathit{Dom}^{\alpha} \to \mathit{Dom}^{\beta}]]$,
- $\mathbf{uapp} = (\mathbf{uapp}_{\mathfrak{T},\varphi} \mid \mathfrak{T} \in \mathfrak{Kind}, \varphi \in \mathbf{T}^{\mathfrak{T}\to\star})$ is a family of continuous functions with $\mathbf{uapp}_{\mathfrak{T},\varphi} : [\mathit{Dom}^{\Pi^{\mathfrak{T}}(\varphi)} \to [\prod_{\alpha \in \mathbf{T}^{\mathfrak{T}}} (\mathit{Dom}^{\varphi(\alpha)})]]$,
- $\mathbf{const} : \mathit{const} \to \mathit{Dom}$ is a mapping function from constants to values such that $\mathbf{const}(c) \in \mathit{Dom}^{\mathcal{T}[\![T]\!]}$ for all $c \in \mathit{const}$ with $T = \text{type } c$.

The applicative structure $\mathcal{E}$ is *extensional* if $\mathbf{app}_{\alpha,\beta}$ and $\mathbf{uapp}_{\mathfrak{T},\varphi}$ are one-to-one.

□

The function Dom assigns a type, that is, a domain, to each element of $\mathbf{T}^{\star}$. In the case of the finitary projection model, Dom^{π} simply is the range of π (recall that the range of a finitary projection is by definition a domain). Perhaps surprisingly, in this model there is even a bijection between $\mathbf{T}^{\star}$ and Dom, that is, $\alpha = \beta \equiv \mathit{Dom}^{\alpha} = \mathit{Dom}^{\beta}$. Thus, each element of $\mathbf{T}^{\star}$ represents a unique type.

An environment η is a mapping $\eta : (\mathit{Var} \to \mathbf{T}) \uplus (\mathit{var} \to \mathit{Dom})$ such that $\eta(A) \in \mathbf{T}^{\mathfrak{T}}$ for all $A \in \mathit{Var}$ with $\mathfrak{T} = \text{kind } A$ and $\eta(a) \in \mathit{Dom}^{\mathcal{T}[\![T]\!]\eta}$ for all $a \in \mathit{var}$ with $T = \text{type } a$.

DEFINITION 2.31 An applicative structure for the polymorphic λ-calculus $\mathcal{E} = (\mathcal{T}, Dom, \mathsf{app}, \mathsf{uapp}, \mathsf{const})$ satisfies the *environment model condition* if the clauses below define a total meaning function, where the meaning function is defined by induction on the structure of typing derivations.

$$
\begin{array}{lcl}
\mathcal{E}[\![t :: T]\!]\eta & \in & Dom^{\mathcal{T}[\![T]\!]\eta} \\
\mathcal{E}[\![c :: T]\!]\eta & = & \mathsf{const}(c) \\
\mathcal{E}[\![a :: T]\!]\eta & = & \eta(a) \\
\mathcal{E}[\![(\lambda a \,.\, t) :: (S \to T)]\!]\eta & = & \text{the unique } \varphi \in Dom^{\mathsf{Const}(\to)\ \sigma\ \tau} \text{ such that} \\
 & & \quad \forall \delta \in Dom^{\sigma} \,.\, \mathsf{app}_{\sigma,\tau}\ \varphi\ \delta = \mathcal{E}[\![t :: T]\!]\eta(a := \delta) \\
 & & \text{where } \sigma = \mathcal{T}[\![S]\!]\eta \text{ and } \tau = \mathcal{T}[\![T]\!]\eta \\
\mathcal{E}[\![(t\ u) :: S]\!]\eta & = & \mathsf{app}_{\upsilon,\omega}\ (\mathcal{E}[\![t :: U \to V]\!]\eta)\ (\mathcal{E}[\![u :: U]\!]\eta) \\
 & & \text{where } \upsilon = \mathcal{T}[\![U]\!]\eta \text{ and } \omega = \mathcal{T}[\![V]\!]\eta \\
\mathcal{E}[\![(\lambda A \,.\, t) :: (\forall A \,.\, T)]\!]\eta & = & \text{the unique } \psi \in Dom^{\Pi^{\mathfrak{U}}(\varphi)} \text{ such that} \\
 & & \quad \forall \alpha \in \mathbf{T}^{\mathfrak{U}} \,.\, \mathsf{uapp}_{\mathfrak{U},\varphi}\ \psi\ \alpha = \mathcal{E}[\![t :: T]\!]\eta(A := \alpha) \\
 & & \text{where } \mathfrak{U} = \text{kind } A \text{ and } \varphi(\alpha) = \mathcal{T}[\![T]\!]\eta(A := \alpha) \\
\mathcal{E}[\![(t\ U) :: V[A := U]]\!]\eta & = & \mathsf{uapp}_{\mathfrak{U},\varphi}\ (\mathcal{E}[\![t :: \forall A \,.\, V]\!]\eta)\ (\mathcal{T}[\![U :: \mathfrak{U}]\!]\eta) \\
 & & \text{where } \mathfrak{U} = \text{kind } A \text{ and } \varphi(\alpha) = \mathcal{T}[\![V]\!]\eta\ (A := \alpha) \\
\mathcal{E}[\![(\mathit{fix}\ t) :: U]\!]\eta & = & \bigsqcup\{\delta_n \mid n \in \mathbb{N}\} \\
 & & \begin{array}{lcl} \text{where } \delta_0 & = & \bot \\ \qquad\quad \delta_{n+1} & = & \mathsf{app}_{\upsilon,\upsilon}\ (\mathcal{E}[\![t :: U \to U]\!]\eta)\ \delta_n \\ \qquad\quad \upsilon & = & \mathcal{T}[\![U]\!]\eta \end{array} \\
\mathcal{E}[\![t :: U]\!]\eta & = & \mathcal{E}[\![t :: T]\!]\eta \\
 & & \text{where } T \approx U.
\end{array}
$$

The applicative structure $\mathcal{E}$ is an *environment model of the polymorphic λ-calculus* if $\mathcal{T}$ is a type model and if $\mathcal{E}$ is extensional and satisfies the environment model condition. □

The definition of the meaning function proceeds by induction on the structure of typing derivations. However, because of rule (CONV) there may be more than one derivation. Fortunately, it is relatively easy to show that the meaning of a well-typed term does not depend on the particular typing derivation we use, the main reason being that structurally equivalent type terms possess the same meaning.

Chapter 3

Generic programs

This chapter constitutes the core of the thesis. It shows how to program generically and how to specialize a given generic definition to concrete instances of data types. In fact, we will get to know two different forms of generic definitions. The first form, called POPL-style, is easier to use from the generic programmer's point of view, whereas the second, called MPC-style[1], is considerably more general. Because the second form builds heavily upon the first, it is necessary to introduce them both.

This chapter is organized as follows. Section 3.1 sets the scene explaining in some detail the definition of generic values such as *encode* or *decodes* that are indexed by types of kind $\star$. Section 3.2 then generalizes the definitional scheme to values such as *size* that are indexed by types of first- or second-order kinds. Section 3.3 generalizes even further and explains how to define values that are indexed by types of arbitrary kinds. Finally, Section 3.4 reviews related work.

3.1 Type-indexed values

Before we start the formal investigation, let us briefly recall the basics of generic programming from the introduction.

A standard example of a generic function is testing two values for equality. We have already remarked that we cannot define a *polymorphic* equality function that has type $\forall\, T.\ T \to T \to \mathit{Bool}$. A polymorphic function is an algorithm that is insensitive to what type the values in some structure are, so a function of type $\forall\, T.\ T \to T \to \mathit{Bool}$ must necessarily be constant—this informal argument can be made precise using the parametricity theorem (Wadler 1989). However, the equality function enjoys a generic definition as it can be defined by induction on the structure of its type argument.

$$
\begin{array}{lcl}
\mathit{equal}\langle T :: \star\rangle & :: & T \to T \to \mathit{Bool} \\
\mathit{equal}\langle 1\rangle\ u_1\ u_2 & = & \mathit{true} \\
\mathit{equal}\langle \mathit{Char}\rangle\ c_1\ c_2 & = & \mathit{equalChar}\ c_1\ c_2 \\
\mathit{equal}\langle \mathit{Int}\rangle\ i_1\ i_2 & = & \mathit{equalInt}\ i_1\ i_2 \\
\mathit{equal}\langle A + B\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inl}\ a_2) & = & \mathit{equal}\langle A\rangle\ a_1\ a_2 \\
\mathit{equal}\langle A + B\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inr}\ b_2) & = & \mathit{false} \\
\mathit{equal}\langle A + B\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inl}\ a_2) & = & \mathit{false} \\
\mathit{equal}\langle A + B\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inr}\ b_2) & = & \mathit{equal}\langle B\rangle\ b_1\ b_2 \\
\mathit{equal}\langle A \times B\rangle\ (a_1, b_1)\ (a_2, b_2) & = & \mathit{equal}\langle A\rangle\ a_1\ a_2 \wedge \mathit{equal}\langle B\rangle\ b_1\ b_2
\end{array}
$$

The type signature makes precise that *equal* is indexed by a type of kind $\star$ and that the type of $\mathit{equal}\langle T\rangle$ depends on T. To define *equal* it suffices to supply

[1] Rather unimaginatively, the two styles are called after the conferences where I first published the respective results: Symposium on Principles of Programming Languages (POPL' 00) and Conference on Mathematics of Program Construction (MPC 2000).

instances for each of the primitive type constructors. Note, however, that *equal* cannot be defined for the function space constructor. Let us consider each equation in turn. Since '1' comprises only one element, two elements of type '1' are trivially equal. For *Char* and *Int* generic equality falls back on the functions *equalChar* and *equalInt* supplied from elsewhere. Elements of a sum type are equal if they have the same constructor and the arguments of the constructor are equal. Finally, two pairs are equal if the corresponding components are equal.

The following sections study generic definitions in detail. Section 3.1.1 characterizes the set of normal forms of types of kind $\star$, Section 3.1.2 introduces the general scheme for defining generic values indexed by types of this kind, and Section 3.1.3 shows how to specialize a generic value thus defined to types of arbitrary kinds.

3.1.1 Normal forms of types

The simple inductive definition of *equal* is quite elegant but does it cover all possible cases? Recall that the type language of Haskell is far more complex including among other things type abstraction and type recursion. Now, it turns out that we have to make one basic assumption, namely, that a generic definition yields the same instance when applied to structurally equivalent types, that is, $equal\langle T_1\rangle = equal\langle T_2\rangle$ if $T_1 \approx T_2$. This is, however, a very reasonable assumption since structurally equivalent types are interchangeable using typing rule (CONV). Given this assumption it is then sufficient to consider as type indices types in normal form where normal form means 'infinite normal form', that is, the Böhm tree of a type.

Working with potentially infinite type terms is not as problematic as one might think at first sight. After all, in a non-strict language such as Haskell we happily operate on potentially infinite objects such as infinite lists or trees. In fact, we will show in the next section how to implement a poor man's version of generic equality in Haskell using infinite type terms.

For the following treatment let us assume that the set of type constants *Const* is given by $Const = \{1, Char, Int, (+), (\times), (\rightarrow)\}$. Note that *Const* only includes zeroth- or first-order kinded type constants, that is, $order(\mathfrak{C}) \leqslant 1$ for all type constants $C :: \mathfrak{C}$. We will see later that this is an essential requirement for POPL-style definitions.

Now, types of kind $\star$ have a very simple normal form. Consider the Böhm tree of a type of kind $\star$. Clearly, the root of the tree cannot be labelled with a type abstraction. Instead, it must be labelled with a primitive type constructor, say, C. Moreover, if C has arity n, the root must have n direct successors (since Böhm trees are based on η-head-normal forms). Thus, the normal form of type terms of kind $\star$ is described by the following grammar.

$$\begin{array}{lcl}
\mathrm{NF}^\star & ::= & 1 \\
& | & \mathit{Char} \\
& | & \mathit{Int} \\
& | & \mathrm{NF}_1^\star + \mathrm{NF}_2^\star \\
& | & \mathrm{NF}_1^\star \times \mathrm{NF}_2^\star \\
& | & \mathrm{NF}_1^\star \rightarrow \mathrm{NF}_2^\star
\end{array}$$

Even though specified by a grammar, it is understood that $\mathrm{NF}^\star$ includes finite and infinite type terms. In particular, we have $\{\mathrm{BT}(T) \mid T :: \star\} \subseteq \mathrm{NF}^\star = \mathcal{B}^\star$.

3.1.2 Defining generic values

While I prefer Haskell for the practical examples, I will use the polymorphic λ-calculus for the theoretical treatment of generic programming. The main reason for this choice is that we require rank-n polymorphism for the specialization of generic values but Haskell only supports rank-1 polymorphism (extensions of Haskell allow for rank-2 type signatures).

The characterization of normal forms motivates the following scheme for defining type-indexed values.

$$
\begin{array}{lcl}
poly\langle T :: \star\rangle & :: & Poly\ T \\
poly\langle 1\rangle & = & poly_1 \\
poly\langle Char\rangle & = & poly_{Char} \\
poly\langle Int\rangle & = & poly_{Int} \\
poly\langle A + B\rangle & = & poly_{+}\ A\ (poly\langle A\rangle)\ B\ (poly\langle B\rangle) \\
poly\langle A \times B\rangle & = & poly_{\times}\ A\ (poly\langle A\rangle)\ B\ (poly\langle B\rangle) \\
poly\langle A \to B\rangle & = & poly_{\to}\ A\ (poly\langle A\rangle)\ B\ (poly\langle B\rangle)
\end{array}
$$

Here, *poly* is the name of the type-indexed value; T, A, and B are type variables of kind $\star$; $Poly$, $poly_1$, $poly_{Char}$, $poly_{Int}$, $poly_{+}$, $poly_{\times}$, and $poly_{\to}$ are the ingredients that have to be supplied by the generic programmer. The type of $poly\langle T\rangle$ is given by $Poly\ T$, where $Poly$ is a type constructor of kind $\star \to \star$. Note that unlike the type index $Poly$ may also contain polymorphic types. The $poly_C$ values must have the following types:

$$
\begin{array}{lcl}
poly_1 & :: & Poly\ 1 \\
poly_{Int} & :: & Poly\ Int \\
poly_{+} & :: & \forall A\,.\,Poly\ A \to \forall B\,.\,Poly\ B \to Poly\ (A + B) \\
poly_{\times} & :: & \forall A\,.\,Poly\ A \to \forall B\,.\,Poly\ B \to Poly\ (A \times B) \\
poly_{\to} & :: & \forall A\,.\,Poly\ A \to \forall B\,.\,Poly\ B \to Poly\ (A \to B).
\end{array}
$$

In the latter three cases A and B are universally quantified since $poly_{+}$, $poly_{\times}$ and $poly_{\to}$ have to work for all possible argument types.

In practice, we do not require that an instance is provided for every type constant C in $Const$. In case an instance for C is missing, we tacitly add $poly_C = undefined$. Alternatively, one can generate a compile-time error if an attempt is made to specialize *poly* for a type that includes C.

It is instructive to see how the example given in the introduction to Section 3.1 maps to the formalism above: $equal\langle T\rangle$ has type $Equal\ T = T \to T \to Bool$ and the functions $equal_1$, $equal_{Char}$, $equal_{Int}$, $equal_{+}$ and $equal_{\times}$ are given by

$$
\begin{array}{lcl}
equal_1 & = & \lambda u_1 :: 1\,.\,\lambda u_2 :: 1\,.\,true \\
equal_{Char} & = & \lambda c_1 :: Char\,.\,\lambda c_2 :: Char\,.\,equalChar\ c_1\ c_2 \\
equal_{Int} & = & \lambda i_1 :: Int\,.\,\lambda i_2 :: Int\,.\,equalInt\ i_1\ i_2 \\
equal_{+} & = & \lambda A\,.\,\lambda equal_A :: (A \to A \to Bool)\,.\,\lambda B\,.\,\lambda equal_B :: (B \to B \to Bool)\,. \\
& & \lambda s_1 :: A + B\,.\,\lambda s_2 :: A + B\,. \\
& & \quad \mathbf{case}\ s_1\ \mathbf{of}\ \{\,inl\ a_1 \Rightarrow \mathbf{case}\ s_2\ \mathbf{of}\ \{\,inl\ a_2 \Rightarrow equal_A\ a_1\ a_2; inr\ b_2 \Rightarrow false\,\}; \\
& & \qquad\qquad\qquad inr\ b_1 \Rightarrow \mathbf{case}\ s_2\ \mathbf{of}\ \{\,inl\ a_2 \Rightarrow false; inr\ b_2 \Rightarrow equal_B\ b_1\ b_2\,\}\,\} \\
equal_{\times} & = & \lambda A\,.\,\lambda equal_A :: (A \to A \to Bool)\,.\,\lambda B\,.\,\lambda equal_B :: (B \to B \to Bool)\,. \\
& & \lambda p_1 :: A \times B\,.\,\lambda p_2 :: A \times B\,. \\
& & \quad equal_A\ (outl\ p_1)\ (outl\ p_2) \wedge equal_B\ (outr\ p_1)\ (outr\ p_2).
\end{array}
$$

The essential difference to the original Haskell code is that universal abstractions and applications are made explicit.

Turning to the semantics of generic definitions let us assume that we are given an environment model $\mathcal{E} = (\mathcal{T}, Dom, \mathbf{app}, \mathbf{uapp}, \mathbf{const})$ for the polymorphic λ-calculus. We will specify the semantics of generic definitions relative to this model. To simplify notation we omit $\mathbf{app}_{T,U}$, $\mathbf{app}^{-1}_{T,U}$, $\mathbf{uapp}_{\mathfrak{T},\varphi}$, $\mathbf{uapp}^{-1}_{\mathfrak{T},\varphi}$ and we abbreviate $\mathcal{T}[\![T]\!]$ by $[\![T]\!]$ and $\mathcal{E}[\![t]\!]$ by $[\![t]\!]$.

The definition of type-indexed values is inductive on the structure of $\mathrm{NF}^\star$: we have one equation for each primitive type constructor. Now, a standard result from the theory of infinite trees (Courcelle 1983) guarantees that every inductively defined function that maps trees to elements of some domain possesses a unique least extension in the realm of infinite trees. Define $\mathbf{poly}\langle T\rangle = [\![poly\langle T\rangle]\!]$ and $\mathbf{poly}_C = [\![poly_C]\!]$, then there exists a *unique least extension* such that $\mathbf{poly}\langle\Omega\rangle = \bot$—that is, $\mathbf{poly}$ is strict—and

$$\mathbf{poly}\langle \mathrm{BT}(C)\ \tau_1\ \cdots\ \tau_n\rangle \quad = \quad \mathbf{poly}_C\ [\![\tau_1]\!]\ (\mathbf{poly}\langle\tau_1\rangle)\ \cdots\ [\![\tau_n]\!]\ (\mathbf{poly}\langle\tau_n\rangle)$$

for all type constants $C :: \star^n \to \star$ and for all Böhm trees $\tau_1, \ldots, \tau_n$. In that sense, *poly* is uniquely defined by its action on primitive type constructors, that is, by $poly_1$, $poly_{Char}$, $poly_{Int}$, $poly_+$, $poly_\times$ and $poly_\to$.

To summarize, the semantics of $poly\langle T\rangle$, where $T \in MonoType$ is a closed monomorphic type term, is given by $\mathbf{poly}\langle \mathrm{BT}(T)\rangle$. Thus, to evaluate $poly\langle T\rangle$ we apply the extension of *poly* to the Böhm tree of T.

Before we proceed let us briefly discuss how to implement generic definitions in Haskell. Since Haskell does not support the definition of values that depend on types, we have to work with encodings of types and a so-called universal data type. Figures 3.1 and 3.2 summarize the implementation.

The data type *Type*, which corresponds to $\mathrm{NF}^\star$, is used to represent types of kind $\star$. Type constructors of kind $\star \to \star$ are simply given by functions of type $Type \to Type$. Since Haskell is a non-strict language, recursive data type declarations can be directly translated into recursive value definitions. The functions *list* and *perfect* serve as examples.

The data type *Univ* is a so-called *universal data type* that can be used to represent values of an arbitrary type formed according to the grammar of $\mathrm{NF}^\star$. The class *EP* then introduces a function for embedding values into the universal data type and a function for projecting values back. Perhaps surprisingly, *embed* and *project* also enjoy generic definitions.

$$\begin{array}{lcl}
embed\langle T :: \star\rangle & :: & T \to Univ \\
embed\langle 1\rangle\ u & = & U1\ u \\
embed\langle Char\rangle\ c & = & UChar\ c \\
embed\langle Int\rangle\ i & = & UInt\ i \\
embed\langle A + B\rangle\ (inl\ a) & = & USum\ (inl\ (embed\langle A\rangle\ a)) \\
embed\langle A + B\rangle\ (inr\ b) & = & USum\ (inr\ (embed\langle B\rangle\ b)) \\
embed\langle A \times B\rangle\ (a, b) & = & UPair\ (embed\langle A\rangle\ a, embed\langle B\rangle\ b) \\
embed\langle A \to B\rangle\ f & = & UFun\ (embed\langle B\rangle \cdot f \cdot project\langle A\rangle) \\
project\langle T :: \star\rangle & :: & Univ \to T \\
project\langle 1\rangle\ (U1\ u) & = & u \\
project\langle Char\rangle\ (UChar\ c) & = & c \\
project\langle Int\rangle\ (UInt\ i) & = & i \\
project\langle A + B\rangle\ (USum\ (inl\ a)) & = & inl\ (project\langle A\rangle\ a) \\
project\langle A + B\rangle\ (USum\ (inr\ b)) & = & inr\ (project\langle B\rangle\ b) \\
project\langle A \times B\rangle\ (UPair\ (a, b)) & = & (project\langle A\rangle\ a, project\langle B\rangle\ b) \\
project\langle A \to B\rangle\ (UFun\ f) & = & project\langle B\rangle \cdot f \cdot embed\langle A\rangle
\end{array}$$

$$
\begin{array}{lcl}
\{\text{- } \mathit{representing\ types} \text{ -}\} & & \\
\mathbf{data}\ \mathit{Type} & = & \mathit{TChar} \\
 & | & \mathit{TInt} \\
 & | & \mathit{T1} \\
 & | & \mathit{Type} :\!+\!: \mathit{Type} \\
 & | & \mathit{Type} :\!\times\!: \mathit{Type} \\
 & | & \mathit{Type} :\!\to\!: \mathit{Type} \\
\mathit{char}, \mathit{int}, \mathit{string} & :: & \mathit{Type} \\
\mathit{char} & = & \mathit{TChar} \\
\mathit{int} & = & \mathit{TInt} \\
\mathit{string} & = & \mathit{list}\ \mathit{char} \\
\mathit{list}, \mathit{perfect} & :: & \mathit{Type} \to \mathit{Type} \\
\mathit{list}\ a & = & \mathit{T1} :\!+\!: (a :\!\times\!: \mathit{list}\ a) \\
\mathit{perfect}\ a & = & a :\!+\!: \mathit{perfect}\ (a :\!\times\!: a) \\
\{\text{- } \mathit{a\ universal\ datatype} \text{ -}\} & & \\
\mathbf{data}\ \mathit{Univ} & = & \mathit{UChar}\ \mathit{Char} \\
 & | & \mathit{UInt}\ \mathit{Int} \\
 & | & \mathit{U1}\ 1 \\
 & | & \mathit{USum}\ (\mathit{Univ} + \mathit{Univ}) \\
 & | & \mathit{UPair}\ (\mathit{Univ} \times \mathit{Univ}) \\
 & | & \mathit{UFun}\ (\mathit{Univ} \to \mathit{Univ}) \\
\mathbf{class}\ \mathit{EP}\ A\ \mathbf{where} & & \\
\quad \mathit{embed} & :: & A \to \mathit{Univ} \\
\quad \mathit{project} & :: & \mathit{Univ} \to A \\
\mathbf{instance}\ \mathit{EP}\ \mathit{Univ}\ \mathbf{where} & & \\
\quad \mathit{embed} & = & \mathit{id} \\
\quad \mathit{project} & = & \mathit{id} \\
\mathbf{instance}\ \mathit{EP}\ \mathit{Char}\ \mathbf{where} & & \\
\quad \mathit{embed}\ c & = & \mathit{UChar}\ c \\
\quad \mathit{project}\ (\mathit{UChar}\ c) & = & c \\
\mathbf{instance}\ \mathit{EP}\ \mathit{Int}\ \mathbf{where} & & \\
\quad \mathit{embed}\ i & = & \mathit{UInt}\ i \\
\quad \mathit{project}\ (\mathit{UInt}\ i) & = & i \\
\mathbf{instance}\ \mathit{EP}\ 1\ \mathbf{where} & & \\
\quad \mathit{embed}\ u & = & \mathit{U1}\ u \\
\quad \mathit{project}\ (\mathit{U1}\ u) & = & u \\
\mathbf{instance}\ (\mathit{EP}\ A, \mathit{EP}\ B) \Rightarrow \mathit{EP}\ (A + B)\ \mathbf{where} & & \\
\quad \mathit{embed}\ (\mathit{inl}\ a) & = & \mathit{USum}\ (\mathit{inl}\ (\mathit{embed}\ a)) \\
\quad \mathit{embed}\ (\mathit{inr}\ b) & = & \mathit{USum}\ (\mathit{inr}\ (\mathit{embed}\ b)) \\
\quad \mathit{project}\ (\mathit{USum}\ (\mathit{inl}\ a)) & = & \mathit{inl}\ (\mathit{project}\ a) \\
\quad \mathit{project}\ (\mathit{USum}\ (\mathit{inr}\ b)) & = & \mathit{inr}\ (\mathit{project}\ b)
\end{array}
$$

Figure 3.1: A poor man's implementation of generic values in Haskell (part 1).

```
instance (EP A, EP B) ⇒ EP (A × B) where
  embed (a, b)                = UPair (embed a, embed b)
  project (UPair (a, b))      = (project a, project b)
instance (EP A, EP B) ⇒ EP (A → B) where
  embed f                     = UFun (embed · f · project)
  project (UFun f)            = project · f · embed
instance (EP A) ⇒ EP [A] where
  embed x                     = embed (case x of {[] → inl (); a : as → inr (a, as)})
  project x                   = case project x of {inl () → []; inr (a, as) → a : as}
instance (EP A) ⇒ EP (Fork A) where
  embed x                     = embed (case x of {fork a₁ a₂ → (a₁, a₂)})
  project x                   = case project x of {(a₁, a₂) → fork a₁ a₂}
instance (EP A) ⇒ EP (Perfect A) where
  embed x                     = embed (case x of {zeroP a → inl a; succP t → inr t})
  project x                   = case project x of {inl a → zeroP a; inr t → succP t}
{- generic equality -}
equal                         :: Type → Univ → Univ → Bool
equal TChar c₁ c₂             = equalChar (project c₁) (project c₂)
equal TInt i₁ i₂              = equalInt (project i₁) (project i₂)
equal T1 u₁ u₂                = true
equal (a :+: b) s₁ s₂         = case (project s₁, project s₂) of
  (inl a₁, inl a₂)            → equal a a₁ a₂
  (inl a₁, inr b₂)            → false
  (inr b₁, inl a₂)            → false
  (inr b₁, inr b₂)            → equal b b₁ b₂
equal (a :×: b) p₁ p₂         = case (project p₁, project p₂) of
  ((a₁, b₁), (a₂, b₂))        → equal a a₁ a₂ ∧ equal b b₁ b₂
{- specializing generic equality -}
equalString                   :: String → String → Bool
equalString s₁ s₂             = equal string (embed s₁) (embed s₂)
equalPerfectInt               :: Perfect Int → Perfect Int → Bool
equalPerfectInt t₁ t₂         = equal (perfect int) (embed t₁) (embed t₂)
```

Figure 3.2: A poor man's implementation of generic values in Haskell (part 2).

Note that *embed* and *project* are mutually recursive and that the equations also cover functional types. These clauses can be directly mapped to instances of EP. Unfortunately, Haskell's class and instance declarations are an imperfect substitute for generic definitions: we have to provide explicit instances for every data type by hand (we provide instance declarations for '[ ]', *Fork* and *Perfect* in Figure 3.2).

Using the types *Type* and *Univ* we can implement a generic value of type $poly\langle T :: \star\rangle :: Poly\ T$ by a Haskell function of type $poly :: Type \rightarrow Poly\ Univ$. The only difference to a generic definition is that at each stage of the recursion we have to project arguments out of the universal data type and embed results into the universal data type.

Finally, if we require a generic value at some specific instance T, we call the Haskell function with T's encoding. Furthermore, we have to embed arguments into the universal data type and project results back.

3.1.3 Specializing generic values

The purpose of a generic value is to be specialized. Before we look at the formal definition let us motivate the key idea. First of all, note that the poor man's implementation given in the previous section is rather inefficient because *poly* interprets its type argument at each stage of the recursion. The type argument is, however, statically known. By specializing $poly\langle T\rangle$ for a given T we remove this interpretative layer. Thus, we can view the following as a very special instance of *partial evaluation*.

In order to specialize $poly\langle T\rangle$, where T is a closed monomorphic type term, we cannot simply unfold the definition of *poly*. To see why consider the following attempt to specialize $poly\langle Perfect\ Int\rangle$ (to improve readability we omit universal applications):

$$\begin{aligned}
& poly\langle Perfect\ Int\rangle \\
= \;& poly\langle Int + Perfect\ (Fork\ Int)\rangle \\
= \;& poly_{+}\ poly_{Int}\ (poly\langle Perfect\ (Fork\ Int)\rangle) \\
= \;& poly_{+}\ poly_{Int}\ (poly\langle Fork\ Int + Perfect\ (Fork^2\ Int)\rangle) \\
= \;& poly_{+}\ poly_{Int}\ (poly_{+}\ (poly\langle Fork\ Int\rangle)\ (poly\langle Perfect\ (Fork^2\ Int)\rangle)) \\
= \;& \ldots
\end{aligned}$$

To define $poly\langle Perfect\ Int\rangle$ we require $poly\langle Perfect\ (Fork^n\ Int)\rangle$ for each natural number $n \geqslant 1$. So if we simply unfold the definition, we will in general not obtain a finite representation of $poly\langle T\rangle$.

The key idea of the specialization is to mimic the structure of types at the value level. For example, $poly\langle Perfect\ Int\rangle$ should be compositionally defined in terms of specializations for the constituent types, say, $poly_{Perfect}$ and $poly_{Int}$. Since *Perfect* is a function on types, $poly_{Perfect}$ is consequently a function operating on generic values. Then the implementation for the type application *Perfect Int* is given by the application of $poly_{Perfect}$ to $poly_{Int}$. In a nutshell, type abstraction is mapped to value abstraction, type application to value application, and type recursion to value recursion. Note that we have already applied this principle in the introduction when giving ad-hoc definitions for *encode* and *decodes*. Recall, for instance, that the encoder for *List* has type $\forall A\,.\,(A \rightarrow Bin) \rightarrow (List\ A \rightarrow Bin)$. It is a function that maps an encoder for the base type A to an encoder for the type $List\ A$.

It is important to note that when we specialize a generic value *poly* to a particular data type T, we must be prepared to specialize *poly* to types of arbitrary kinds. The reason is simply that the definition of T may involve arbitrarily complex types. For clarity, let us denote the generalization of *poly* that works for types of arbitrary kinds by $poly\langle\!\langle - \rangle\!\rangle$. We call $poly\langle\!\langle - \rangle\!\rangle$ the *promoted* version of *poly*. In general, we reserve single angle brackets for type arguments that range over type of one fixed kind and use double angle brackets for type arguments of arbitrary kinds. The double angle brackets are reminiscent of the semantic brackets $[\![-]\!]$. In fact, we will see shortly that this correspondence is intentional.

Now, since $poly_{Perfect}$ is a function that operates on generic values, it has a type different from $poly_{Int}$. In fact, the type of $poly\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle$ is given by $Poly\langle\mathfrak{T}\rangle\ T$ where $Poly\langle\mathfrak{T}\rangle$ is defined by induction on the structure of kinds.

$$
\begin{array}{lcl}
Poly\langle\mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \star \\
Poly\langle\star\rangle\ T & = & Poly\ T \\
Poly\langle\mathfrak{A} \times \mathfrak{B}\rangle\ T & = & Poly\langle\mathfrak{A}\rangle\ (Outl\ T) \times Poly\langle\mathfrak{B}\rangle\ (Outr\ T) \\
Poly\langle\mathfrak{A} \to \mathfrak{B}\rangle\ T & = & \forall A\,.\,Poly\langle\mathfrak{A}\rangle\ A \to Poly\langle\mathfrak{B}\rangle\ (T\ A)
\end{array}
$$

If T is a pair of types, then $poly\langle\!\langle T \rangle\!\rangle$ is a pair of generic values. Similarly, if T is a type constructor of kind $\mathfrak{A} \to \mathfrak{B}$, then $poly\langle\!\langle T \rangle\!\rangle$ is a function that maps values of type $Poly\langle\mathfrak{A}\rangle\ A$ to values of type $Poly\langle\mathfrak{B}\rangle\ (T\ A)$, for all types A. Again, it is important that A is universally quantified since T may be applied to different types.

The nesting of universal quantifiers is dictated by the kind: if $\mathfrak{T}$ has order n, then $Poly\langle\mathfrak{T}\rangle\ T$ is a rank-n type—assuming that $Poly\ T$ has rank 0. For instance, for $GRose :: (\star \to \star) \to (\star \to \star)$ we have

$$
\begin{array}{cl}
 & Poly\langle(\star \to \star) \to (\star \to \star)\rangle\ GRose \\
= & \forall F\,.\,Poly\langle\star \to \star\rangle\ F \to Poly\langle\star \to \star\rangle\ (GRose\ F) \\
= & \forall F\,.\,(\forall B\,.\,Poly\langle\star\rangle\ B \to Poly\langle\star\rangle\ (F\ B)) \to (\forall A\,.\,Poly\langle\star\rangle\ A \to Poly\langle\star\rangle\ (GRose\ F\ A)) \\
= & \forall F\,.\,(\forall B\,.\,Poly\ B \to Poly\ (F\ B)) \to (\forall A\,.\,Poly\ A \to Poly\ (GRose\ F\ A)).
\end{array}
$$

Since $GRose$ has an order-2 kind, $Poly\langle\mathfrak{T}\rangle\ GRose$ is a rank-2 type.

The definition of $poly\langle\!\langle T \rangle\!\rangle$ is inductive on the structure of kinding derivations. In fact, we can view the definition as an interpretation of the simply typed λ-calculus.

$$
\begin{array}{lcl}
poly\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & Poly\langle\mathfrak{T}\rangle\ T \\
poly\langle\!\langle C :: \mathfrak{C} \rangle\!\rangle & = & poly_C \\
poly\langle\!\langle A :: \mathfrak{A} \rangle\!\rangle & = & poly_A \\
poly\langle\!\langle (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle & = & (poly\langle\!\langle T_1 :: \mathfrak{T}_1 \rangle\!\rangle, poly\langle\!\langle T_2 :: \mathfrak{T}_2 \rangle\!\rangle) \\
poly\langle\!\langle Outl\ T :: \mathfrak{T}_1 \rangle\!\rangle & = & outl\ (poly\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
poly\langle\!\langle Outr\ T :: \mathfrak{T}_2 \rangle\!\rangle & = & outr\ (poly\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
poly\langle\!\langle (\Lambda A\,.\,T) :: (\mathfrak{V} \to \mathfrak{T}) \rangle\!\rangle & = & \lambda A\,.\,\lambda poly_A\,.\,poly\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle \\
poly\langle\!\langle T\ U :: \mathfrak{V} \rangle\!\rangle & = & (poly\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle)\ U\ (poly\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle) \\
poly\langle\!\langle Fix\ T :: \mathfrak{U} \rangle\!\rangle & = & fix\ ((poly\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle)\ (Fix\ T))
\end{array}
$$

Three remarks are in order. First, we allow only monomorphic types as type indices. This restriction is, however, quite mild. Haskell, for instance, does not allow universal quantifiers in **data** declarations.

Second, for the translation we use a simple variable naming convention, which obviates the need for an explicit environment. We assume that $poly\langle\!\langle A \rangle\!\rangle$ is mapped to the variable $poly_A$, which has type $Poly\langle\mathfrak{A}\rangle\ A$ with $\mathfrak{A} = \text{kind}\ A$. We often write $poly_A$ by concatenating the name of the generic value and the name of the type variable as in *encodeList* or *encodeA*. Of course, to avoid name capture we assume that $poly_A$ is distinct from variables introduced by the generic programmer.

Third, the last equation of the definition probably requires some explanation. The instance $poly\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle$ has type $\forall A\,.\,Poly\langle\mathfrak{U}\rangle\ A \to Poly\langle\mathfrak{U}\rangle\ (T\ A)$. Supplying $Fix\ T$ as the type argument and noting that $T\ (Fix\ T) \approx T$, we obtain a value of type $Poly\langle\mathfrak{U}\rangle\ (Fix\ T)$ as desired.

Remark 3.1 The structure of $poly\langle\!\langle T\rangle\!\rangle$'s definition becomes more visible if we omit kind annotations, universal abstractions and universal applications.

$$
\begin{array}{lcl}
poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & Poly\langle\mathfrak{T}\rangle\ T \\
poly\langle\!\langle C\rangle\!\rangle & = & poly_C \\
poly\langle\!\langle A\rangle\!\rangle & = & poly_A \\
poly\langle\!\langle (T_1, T_2)\rangle\!\rangle & = & (poly\langle\!\langle T_1\rangle\!\rangle, poly\langle\!\langle T_2\rangle\!\rangle) \\
poly\langle\!\langle Outl\ T\rangle\!\rangle & = & outl\ (poly\langle\!\langle T\rangle\!\rangle) \\
poly\langle\!\langle Outr\ T\rangle\!\rangle & = & outr\ (poly\langle\!\langle T\rangle\!\rangle) \\
poly\langle\!\langle \Lambda A\,.\,T\rangle\!\rangle & = & \lambda poly_A\,.\,poly\langle\!\langle T\rangle\!\rangle \\
poly\langle\!\langle T\ U\rangle\!\rangle & = & (poly\langle\!\langle T\rangle\!\rangle)\ (poly\langle\!\langle U\rangle\!\rangle) \\
poly\langle\!\langle Fix\ T\rangle\!\rangle & = & fix\ (poly\langle\!\langle T\rangle\!\rangle)
\end{array}
$$

Indeed, type abstraction is mapped to value abstraction, type application to value application, and type recursion to value recursion. □

Now, the specialized version of $poly\langle T\rangle$, which we write $poly_T$, is simply

$$poly_T \quad = \quad poly\langle\!\langle T\rangle\!\rangle.$$

As an example, the specialized version of $poly\langle Perfect\ Int\rangle$ is $poly_{Perfect}\ Int\ poly_{Int}$ where $poly_{Perfect}$ is given by

$$
\begin{array}{lcl}
poly_{Fork} & :: & \forall A\,.\,Poly\ A \to Poly\ (Fork\ A) \\
poly_{Fork} & = & \lambda A\,.\,\lambda poly_A\,.\,poly_\times\ A\ poly_A\ A\ poly_A \\
poly_{Perfect} & :: & \forall A\,.\,Poly\ A \to Poly\ (Perfect\ A) \\
poly_{Perfect} & = & fix\ ((\lambda P\,.\,\lambda poly_P :: (\forall B\,.\,Poly\ B \to Poly\ (P\ B))\,.\,\lambda A\,.\,\lambda poly_A :: Poly\ A\,. \\
& & \quad poly_+\ A\ poly_A\ (P\ (Fork\ A))\ (poly_P\ (Fork\ A)\ (poly_{Fork}\ A\ poly_A)))\ Perfect).
\end{array}
$$

We can simplify the last definition slightly by performing a λ-reduction and by writing $a = fix\ f$ as the recursive equation $a = f\ a$.

$$
\begin{array}{lcl}
poly_{Perfect} & :: & \forall A\,.\,Poly\ A \to Poly\ (Perfect\ A) \\
poly_{Perfect}\ A\ poly_A & = & poly_+\ A\ poly_A\ (Perfect\ (Fork\ A)) \\
& & \qquad (poly_{Perfect}\ (Fork\ A)\ (poly_{Fork}\ A\ poly_A))
\end{array}
$$

The code nicely illustrates why we require polymorphic recursion when we translate it to Haskell: the recursive call is used at an instance, *Fork A*, of the declared type.

As a second example, consider the specialization of *poly* to the ubiquitous list data type $List = \Lambda A\,.\,Fix\ (\Lambda B\,.\,1 + A \times B)$—this is the λ-dropped variant of *List*, see Section 2.5.1.

$$
\begin{array}{lcl}
poly_{List} & :: & \forall A\,.\,Poly\ A \to Poly\ (List\ A) \\
poly_{List} & = & \lambda A\,.\,\lambda poly_A :: Poly\ A\,.\,fix\ ((\lambda L\,.\,\lambda poly_L :: Poly\ L\,. \\
& & poly_+\ 1\ poly_1\ (A \times L)\ (poly_\times\ A\ poly_A\ L\ poly_L))\ (List\ A))
\end{array}
$$

Again, we can simplify the definition slightly, this time by using a local definition.

$$\begin{array}{lcl}
\mathit{poly}_{List} & :: & \forall A\,.\,\mathit{Poly}\ A \to \mathit{Poly}\ (\mathit{List}\ A)\\
\mathit{poly}_{List}\ A\ \mathit{poly}_A & = & \mathit{poly}_L\\
\quad\textbf{where}\ \mathit{poly}_L & :: & \mathit{Poly}\ (\mathit{List}\ A)\\
\qquad\quad \mathit{poly}_L & = & \mathit{poly}_+\ 1\ \mathit{poly}_1\ (A \times \mathit{List}\ A)\ (\mathit{poly}_\times\ A\ \mathit{poly}_A\ (\mathit{List}\ A)\ \mathit{poly}_L)
\end{array}$$

This time ordinary recursion will do (poly_L has not even a polymorphic type).

Finally, let us consider some instances given in Haskell. Figures 3.3 and 3.4 list the specialization of *encode*, defined in Section 1.1.1, to some of the data types introduced in Section 2.1. Note that we have simplified the code by inlining the encode_C instances. The specializations illustrate several interesting points. As to be expected, the function *encodeSequ* makes use of polymorphic recursion: the recursive call has type $\forall A\,.\,(\mathit{Fork}\ A \to \mathit{Bin}) \to (\mathit{Sequ}\ (\mathit{Fork}\ A) \to \mathit{Bin})$, which is a substitution instance of the declared type. In general, polymorphic recursion is required whenever the type recursion is nested. Several functions have rank-2 type signatures; *encodeFMapFork* shows in a nutshell why this is necessary: the argument *encodeFA* is applied at two different instances: the inner call has type $\forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{FA}\ A \to \mathit{Bin})$ while the outer call has type $\forall A\,.\,(\mathit{FA}\ A \to \mathit{Bin}) \to (\mathit{FA}\ (\mathit{FA}\ A) \to \mathit{Bin})$. The functions *encodeFMapSequ* and *encodeSquare′* even combine polymorphic recursion and the specialized use of a polymorphic argument.

The following theorem states that $\mathit{poly}\langle\!\langle - \rangle\!\rangle$ is well-typed.

Theorem 3.2 If $\mathit{poly}\langle\!\langle C :: \mathfrak{C} \rangle\!\rangle :: \mathit{Poly}\langle\mathfrak{C}\rangle\ C$ for all type constants $C \in \mathit{Const}$, then $\mathit{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle :: \mathit{Poly}\langle\mathfrak{T}\rangle\ T$ for all closed monomorphic type terms $T \in \mathit{MonoType}$.

Proof. This is a simple instance of Theorem 3.10. □

The rest of this section is concerned with the proof of correctness. You may wish to skip the following on first reading. Roughly speaking, we have to show that $\mathit{poly}\langle T\rangle = \mathit{poly}\langle\!\langle T \rangle\!\rangle$, that is, the extension of *poly* is equal to the promoted version. The proof takes place in a semantic setting and is based on a variant of logical relations. Here is a brief outline of the proof:

Let $\mathsf{poly}\langle\!\langle T \rangle\!\rangle\eta = [\![\mathit{poly}\langle\!\langle T \rangle\!\rangle]\!]\eta$ be the semantic pendant of $\mathit{poly}\langle\!\langle - \rangle\!\rangle$. We have already mentioned that $\mathsf{poly}\langle\!\langle - \rangle\!\rangle$ can be seen as specifying an interpretation of type terms (or of terms of the simply typed λ-calculus if you like). A second interpretation of type terms is given by the Böhm-tree model. Now, let T be a closed monomorphic type term of kind $\star$. Then we can prove that the Böhm-tree $\tau = \mathrm{BT}(T)$ and the instance $\varphi = \mathsf{poly}\langle\!\langle T \rangle\!\rangle$ are related by $\mathsf{poly}\langle\tau\rangle = \varphi$. This result immediately implies $\mathsf{poly}\langle\mathrm{BT}(T)\rangle = \mathsf{poly}\langle\!\langle T \rangle\!\rangle$ as desired.

For ease of reference here is the definition of $\mathsf{poly}\langle\!\langle - \rangle\!\rangle$ spelled out in detail.

$$\begin{array}{lcl}
\mathsf{poly}\langle\!\langle C :: \mathfrak{C} \rangle\!\rangle\eta & = & \mathsf{poly}_C\\
\mathsf{poly}\langle\!\langle A :: \mathfrak{A} \rangle\!\rangle\eta & = & \eta(\mathit{poly}_A)\\
\mathsf{poly}\langle\!\langle (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle\eta & = & (\mathsf{poly}\langle\!\langle T_1 :: \mathfrak{T}_1 \rangle\!\rangle\eta, \mathsf{poly}\langle\!\langle T_2 :: \mathfrak{T}_2 \rangle\!\rangle\eta)\\
\mathsf{poly}\langle\!\langle \mathit{Outl}\ T :: \mathfrak{T}_1 \rangle\!\rangle\eta & = & \mathsf{outl}\ (\mathsf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle\eta)\\
\mathsf{poly}\langle\!\langle \mathit{Outr}\ T :: \mathfrak{T}_2 \rangle\!\rangle\eta & = & \mathsf{outr}\ (\mathsf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle\eta)\\
\mathsf{poly}\langle\!\langle (\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T} \rangle\!\rangle\eta & = & \boldsymbol{\lambda}\alpha\,.\,\boldsymbol{\lambda}\varphi\,.\,\mathsf{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle\eta(A := \alpha, \mathit{poly}_A := \varphi)\\
\mathsf{poly}\langle\!\langle T\ U :: \mathfrak{V} \rangle\!\rangle\eta & = & (\mathsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle\eta)\ ([\![U]\!]\eta)\ (\mathsf{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle\eta)\\
\mathsf{poly}\langle\!\langle \mathit{Fix}\ T :: \mathfrak{U} \rangle\!\rangle\eta & = & \mathsf{lfp}\ ((\mathsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta)\ (\mathsf{lfp}\ ([\![T]\!]\eta)))
\end{array}$$

$$
\begin{array}{lcl}
\mathit{encodeMaybe} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Maybe}\ A \to \mathit{Bin}) \\
\mathit{encodeMaybe}\ \mathit{encodeA}\ \mathit{nothing} & = & [\mathtt{0}] \\
\mathit{encodeMaybe}\ \mathit{encodeA}\ (\mathit{just}\ a) & = & \mathtt{1} : \mathit{encodeA}\ a \\
\mathit{encodeList} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{List}\ A \to \mathit{Bin}) \\
\mathit{encodeList}\ \mathit{encodeA} & = & \mathit{encodeL} \\
\quad \mathbf{where}\ \mathit{encodeL}\ \mathit{nil} & = & [\mathtt{0}] \\
\qquad\qquad \mathit{encodeL}\ (\mathit{cons}\ a\ \mathit{as}) & = & \mathtt{1} : \mathit{encodeA}\ a +\!\!+\ \mathit{encodeL}\ \mathit{as} \\
\mathit{encodeRose} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Rose}\ A \to \mathit{Bin}) \\
\mathit{encodeRose}\ \mathit{encodeA} & = & \mathit{encodeR} \\
\quad \mathbf{where}\ \mathit{encodeR}\ (\mathit{branch}\ a\ \mathit{ts}) & = & \mathit{encodeA}\ a +\!\!+\ \mathit{encodeList}\ \mathit{encodeR}\ \mathit{ts} \\
\mathit{encodeGRose} & :: & \forall F\,.\,(\forall B\,.\,(B \to \mathit{Bin}) \to (F\ B \to \mathit{Bin})) \\
 & & \qquad \to (\forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{GRose}\ F\ A \to \mathit{Bin})) \\
\mathit{encodeGRose}\ \mathit{encodeF}\ \mathit{encodeA} & = & \mathit{encodeG} \\
\quad \mathbf{where}\ \mathit{encodeG}\ (\mathit{gbranch}\ a\ \mathit{ts}) & = & \mathit{encodeA}\ a +\!\!+\ \mathit{encodeF}\ \mathit{encodeG}\ \mathit{ts} \\
\mathit{encodeFix} & :: & \forall F\,.\,(\forall A\,.\,(A \to \mathit{Bin}) \to (F\ A \to \mathit{Bin})) \\
 & & \qquad \to (\mathit{Fix}\ F \to \mathit{Bin}) \\
\mathit{encodeFix}\ \mathit{encodeF} & = & \mathit{encodeR} \\
\quad \mathbf{where}\ \mathit{encodeR}\ (\mathit{in}\ x) & = & \mathit{encodeF}\ \mathit{encodeR}\ x \\
\mathit{encodeListBase} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\forall B\,.\,(B \to \mathit{Bin}) \\
 & & \qquad\qquad\qquad\qquad \to (\mathit{ListBase}\ A\ B \to \mathit{Bin})) \\
\multicolumn{3}{l}{\mathit{encodeListBase}\ \mathit{encodeA}\ \mathit{encodeB}\ \mathit{nilL}} \\
 & = & [\mathtt{0}] \\
\multicolumn{3}{l}{\mathit{encodeListBase}\ \mathit{encodeA}\ \mathit{encodeB}\ (\mathit{consL}\ a\ b)} \\
 & = & \mathtt{1} : \mathit{encodeA}\ a +\!\!+\ \mathit{encodeB}\ b \\
\mathit{encodeFork} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Fork}\ A \to \mathit{Bin}) \\
\mathit{encodeFork}\ \mathit{encodeA}\ (\mathit{fork}\ a_1\ a_2) & = & \mathit{encodeA}\ a_1 +\!\!+\ \mathit{encodeA}\ a_2 \\
\mathit{encodeSequ} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Sequ}\ A \to \mathit{Bin}) \\
\mathit{encodeSequ}\ \mathit{encodeA}\ \mathit{endS} & = & [\mathtt{0}] \\
\mathit{encodeSequ}\ \mathit{encodeA}\ (\mathit{zeroS}\ s) & = & \mathtt{1} : \mathtt{0} : \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{encodeA})\ s \\
\mathit{encodeSequ}\ \mathit{encodeA}\ (\mathit{oneS}\ a\ s) & = & \mathtt{1} : \mathtt{1} : \mathit{encodeA}\ a +\!\!+\ \mathit{encodeSequ}\ (\mathit{encodeFork}\ \mathit{encodeA})\ s
\end{array}
$$

Figure 3.3: Specializing *encode* to different data types (part 1).

$$
\begin{array}{lcl}
\mathit{encodeFMapFork} & :: & \forall FA\,.\,(\forall W\,.\,(W \to \mathit{Bin}) \to (FA\ W \to \mathit{Bin})) \\
& & \qquad \to (\forall V\,.\,(V \to \mathit{Bin}) \to (\mathit{FMapFork}\ FA\ V \to \mathit{Bin})) \\
\multicolumn{3}{l}{\mathit{encodeFMapFork}\ \mathit{encodeFA}\ \mathit{encodeV}\ (\mathit{trieFork}\ \mathit{tf})} \\
& = & \mathit{encodeFA}\ (\mathit{encodeFA}\ \mathit{encodeV})\ \mathit{tf} \\
\mathit{encodeFMapSequ} & :: & \forall FA\,.\,(\forall W\,.\,(W \to \mathit{Bin}) \to (FA\ W \to \mathit{Bin})) \\
& & \qquad \to (\forall V\,.\,(V \to \mathit{Bin}) \to (\mathit{FMapSequ}\ FA\ V \to \mathit{Bin})) \\
\multicolumn{3}{l}{\mathit{encodeFMapSequ}\ \mathit{encodeFA}\ \mathit{encodeV}\ \mathit{nullSequ}} \\
& = & [\mathtt{0}] \\
\multicolumn{3}{l}{\mathit{encodeFMapSequ}\ \mathit{encodeFA}\ \mathit{encodeV}\ (\mathit{trieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to})} \\
& = & \mathtt{1} : \mathit{encodeMaybe}\ \mathit{encodeV}\ \mathit{te} \\
& +\!\!+ & \mathit{encodeFMapSequ}\ (\mathit{encodeFMapFork}\ \mathit{encodeFA})\ \mathit{encodeV}\ \mathit{tz} \\
& +\!\!+ & \mathit{encodeFA}\ (\mathit{encodeFMapSequ} \\
& & \qquad\qquad (\mathit{encodeFMapFork}\ \mathit{encodeFA})\ \mathit{encodeV})\ \mathit{to} \\
\mathit{encodeSquare} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Square}\ A \to \mathit{Bin}) \\
\mathit{encodeSquare}\ \mathit{encodeA}\ m & = & \mathit{encodeSquare}'\ \mathit{encodeNil}\ \mathit{encodeA}\ m \\
\mathit{encodeSquare}' & :: & \forall F\,.\,(\forall B\,.\,(B \to \mathit{Bin}) \to (F\ B \to \mathit{Bin})) \\
& & \qquad \to (\forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Square}'\ F\ A \to \mathit{Bin})) \\
\multicolumn{3}{l}{\mathit{encodeSquare}'\ \mathit{encodeF}\ \mathit{encodeA}\ (\mathit{zeroM}\ m)} \\
& = & \mathtt{0} : \mathit{encodeF}\ (\mathit{encodeF}\ \mathit{encodeA})\ m \\
\multicolumn{3}{l}{\mathit{encodeSquare}'\ \mathit{encodeF}\ \mathit{encodeA}\ (\mathit{succM}\ m)} \\
& = & \mathtt{1} : \mathit{encodeSquare}'\ (\mathit{encodeCons}\ \mathit{encodeF})\ \mathit{encodeA}\ m \\
\mathit{encodeNil} & :: & \forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Nil}\ A \to \mathit{Bin}) \\
\mathit{encodeNil}\ \mathit{encodeA}\ \mathit{nilN} & = & [\,] \\
\mathit{encodeCons} & :: & \forall F\,.\,(\forall B\,.\,(B \to \mathit{Bin}) \to (F\ B \to \mathit{Bin})) \\
& & \qquad \to (\forall A\,.\,(A \to \mathit{Bin}) \to (\mathit{Cons}\ F\ A \to \mathit{Bin})) \\
\multicolumn{3}{l}{\mathit{encodeCons}\ \mathit{encodeF}\ \mathit{encodeA}\ (\mathit{consC}\ a\ x)} \\
& = & \mathit{encodeA}\ a +\!\!+ \mathit{encodeF}\ \mathit{encodeA}\ x
\end{array}
$$

Figure 3.4: Specializing *encode* to different data types (part 2).

Before we proceed let us make one small amendment to the definition of $\mathbf{poly}\langle\!\langle - \rangle\!\rangle$, which will simplify the proof of correctness. Consider the last equation, which is concerned with the interpretation of type recursion, and note the nesting of fixed points. Let $\varphi = \mathbf{poly}\langle\!\langle T \rangle\!\rangle \eta$ and $\Phi = [\![T]\!]\eta$, then the right-hand side is $\mathbf{lfp}\ (\varphi\ (\mathbf{lfp}\ \Phi))$. Perhaps surprisingly, we can rewrite this expression using a fixed point operator that works on Φ and φ simultaneously. Let

$$\begin{array}{lclcl} \mathbf{slfp}\ \Phi\ \varphi & = & \bigsqcup\{\delta_n \mid n \in \mathbb{N}\} \\ \text{where } \alpha_0 & = & \bot & \delta_0 & = & \bot \\ \quad\alpha_{n+1} & = & \Phi\ \alpha_n & \delta_{n+1} & = & \varphi\ \alpha_n\ \delta_n, \end{array}$$

then one can prove that $\mathbf{slfp}\ \Phi\ \varphi = \mathbf{lfp}\ (\varphi\ (\mathbf{lfp}\ \Phi))$. In fact, this a simple instance of a more general result due to Bekič, which shows how to solve simultaneous fixed point equations using iterated fixed points (Plotkin 1983). Using $\mathbf{slfp}$ the last equation of $\mathbf{poly}\langle\!\langle - \rangle\!\rangle$ reads

$$\mathbf{poly}\langle\!\langle \mathit{Fix}\ T :: \mathfrak{U} \rangle\!\rangle \eta \quad = \quad \mathbf{slfp}\ ([\![T]\!]\eta)\ (\mathbf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle \eta).$$

Remark 3.3 We can also make this change on the syntactical level. To this end we introduce a family of kind-indexed fixed point operators of type

$$\mathit{sfix}_{\mathfrak{U}} \quad :: \quad \forall F\,.\,(\forall A\,.\,\mathit{Poly}\langle\mathfrak{U}\rangle\ A \to \mathit{Poly}\langle\mathfrak{U}\rangle\ (F\ A)) \to \mathit{Poly}\langle\mathfrak{U}\rangle\ (\mathit{Fix}\ F)$$

and replace the last equation of $\mathit{poly}\langle\!\langle - \rangle\!\rangle$ by

$$\mathit{poly}\langle\!\langle \mathit{Fix}\ T :: \mathfrak{U} \rangle\!\rangle \quad = \quad \mathit{sfix}_{\mathfrak{U}}\ T\ (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle).$$

Whether this has any practical advantages remains to be seen. □

We have already remarked that the proof of correctness is based on a variant of logical relations that relates Böhm-like trees to generic instances. Since the type of a generic instance depends on the type argument, the relation is a subset of a dependent product:

$$\mathcal{S}^{\mathfrak{T}} \subseteq \textstyle\sum \tau \in \mathcal{B}^{\mathfrak{T}}\,.\,\mathit{Dom}\ (\mathbf{Poly}^{\mathfrak{T}}([\![\tau]\!])),$$

where $\mathbf{Poly}^{\mathfrak{A}} = [\![\mathit{Poly}\langle\mathfrak{A}\rangle]\!]$. The members of $\mathcal{S}$ are given by

$$\begin{array}{lcl} (\tau,\varphi) \in \mathcal{S}^{\star} & \equiv & \mathbf{poly}\langle\tau\rangle = \varphi \\ (\tau,\varphi) \in \mathcal{S}^{\mathfrak{T}\times\mathfrak{U}} & \equiv & (\mathbf{outl}\ \tau, \mathbf{outl}\ \varphi) \in \mathcal{S}^{\mathfrak{T}}\ \cap\ (\mathbf{outr}\ \tau, \mathbf{outr}\ \varphi) \in \mathcal{S}^{\mathfrak{U}} \\ (\tau,\varphi) \in \mathcal{S}^{\mathfrak{T}\to\mathfrak{U}} & \equiv & \forall \upsilon \in \mathcal{B}^{\mathfrak{T}}\,.\,\forall \delta \in \mathit{Dom}\ (\mathbf{Poly}^{\mathfrak{T}}([\![\upsilon]\!]))\,. \\ & & \qquad (\upsilon,\delta) \in \mathcal{S}^{\mathfrak{T}}\ \supset\ (\tau\ \upsilon, \varphi\ [\![\upsilon]\!]\ \delta) \in \mathcal{S}^{\mathfrak{U}}. \end{array}$$

Note that $\mathcal{S}$ closely adheres to the structure of logical relations: pairs are related iff the corresponding components are related; functions are related iff related arguments are taken to related results. The only difference to 'classical' logical relations is that in the last clause φ additionally takes $[\![\upsilon]\!]$ as an argument. This is because the instance $\mathit{poly}\langle\!\langle F :: \mathfrak{T} \to \mathfrak{U} \rangle\!\rangle$ is given by a polymorphic function.

Recall from Section 2.4.3 that the set of all Böhm-like trees gives rise to a model of the simply typed λ-calculus. The following fact is a restatement of Fact 2.28.

Fact 3.4 Let V be a monomorphic type term of kind $\mathfrak{V}$. Let $\varrho : \mathit{Var} \to \mathcal{B}$ and $\eta :: \mathit{Var} \to \mathbf{T}$ be two environments such that $[\![\varrho(A)]\!] = \eta(A)$ for every free variable A of V, then

$$[\![\mathcal{B}[\![V]\!]\varrho]\!] = [\![V]\!]\eta. \qquad \square$$

The following Lemma is a variant of Lemma 2.20 suitably modified to work with the relation $\mathcal{S}$.

LEMMA 3.5 Let V be a monomorphic type term of kind $\mathfrak{V}$. Let $\varrho : \mathit{Var} \to \mathcal{B}$ and $\eta :: (\mathit{Var} \to \mathbf{T}) \uplus (\mathit{var} \to \mathit{Dom})$ be two environments such that $[\![\varrho(A)]\!] = \eta(A)$ and $(\varrho(A), \eta(\mathit{poly}_A)) \in \mathcal{S}^{\mathfrak{A}}$ for every free variable $A :: \mathfrak{A}$ of $V :: \mathfrak{V}$, then

$$(\mathcal{B}[\![V :: \mathfrak{V}]\!]\varrho, \mathbf{poly}\langle\!\langle V :: \mathfrak{V}\rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{V}}.$$

PROOF. We proceed by induction on the kinding derivation of $V :: \mathfrak{V}$.

- **Case** $V = C :: \mathfrak{C}$: if $\mathfrak{T} = \star^k \to \star$, then $(\mathrm{BT}(C), \mathbf{poly}\langle\!\langle C\rangle\!\rangle) \in \mathcal{S}^{\mathfrak{T}}$ equals

  $$\mathbf{poly}\langle \mathrm{BT}(C)\ \upsilon_1\ \cdots\ \upsilon_k\rangle = \mathbf{poly}_C\ [\![\upsilon_1]\!]\ (\mathbf{poly}\langle\upsilon_1\rangle)\ \ldots\ [\![\upsilon_k]\!]\ (\mathbf{poly}\langle\upsilon_k\rangle),$$

  which holds by assumption (see Section 3.1.2).

- **Case** $V = A :: \mathfrak{A}$: holds since $\varrho(A)$ and $\eta(\mathit{poly}_A)$ are related.

- **Case** $V = (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2$: by the induction hypothesis we have

  $$(\mathcal{B}[\![T_1 :: \mathfrak{T}_1]\!]\varrho, \mathbf{poly}\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{T}_1} \ \cap\ (\mathcal{B}[\![T_2 :: \mathfrak{T}_2]\!]\varrho, \mathbf{poly}\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{T}_2},$$

  which immediately implies

  $$((\mathcal{B}[\![T_1 :: \mathfrak{T}_1]\!]\varrho, \mathcal{B}[\![T_2 :: \mathfrak{T}_2]\!]\varrho), (\mathbf{poly}\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta, \mathbf{poly}\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta)) \in \mathcal{S}^{\mathfrak{T}_1 \times \mathfrak{T}_2}.$$

- **Case** $V = \mathit{Outl}\ T :: \mathfrak{T}_1$: by the induction hypothesis we have

  $$(\mathcal{B}[\![T :: \mathfrak{T}_1 \times \mathfrak{T}_2]\!]\varrho, \mathbf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{T}_1 \times \mathfrak{T}_2},$$

  which immediately implies

  $$(\mathbf{outl}\ (\mathcal{B}[\![T :: \mathfrak{T}_1 \times \mathfrak{T}_2]\!]\varrho), \mathbf{outl}\ (\mathbf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta)) \in \mathcal{S}^{\mathfrak{T}_1}.$$

- **Case** $V = \mathit{Outr}\ T :: \mathfrak{T}_2$: analogous.

- **Case** $V = (\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}$: We have to show that

$$\begin{aligned}
&(\mathcal{B}[\![(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}]\!]\varrho, \mathbf{poly}\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{S} \to \mathfrak{T}}\\
&\quad \equiv \forall \upsilon\ \delta\,.\,(\upsilon, \delta) \in \mathcal{S}^{\mathfrak{S}}\\
&\qquad\qquad \supset (\mathcal{B}[\![(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}]\!]\varrho\ \upsilon, \mathbf{poly}\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta\ [\![\upsilon]\!]\ \delta) \in \mathcal{S}^{\mathfrak{T}}
\end{aligned}$$

  Assume that $(\upsilon, \delta) \in \mathcal{S}^{\mathfrak{S}}$. Since the modified environments $\varrho(A := \upsilon)$ and $\eta(A := [\![\upsilon]\!], \mathit{poly}_A := \delta)$ are related, we can invoke the induction hypothesis to obtain

  $$(\mathcal{B}[\![T :: \mathfrak{T}]\!]\varrho(A := \upsilon), \mathbf{poly}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta(A := [\![\upsilon]\!], \mathit{poly}_A := \delta)) \in \mathcal{S}^{\mathfrak{T}}.$$

  Now, since $\mathcal{B}[\![(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}]\!]\varrho\ \upsilon = \mathcal{B}[\![T :: \mathfrak{T}]\!]\varrho(A := \upsilon)$ and furthermore $\mathbf{poly}\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta\ [\![\upsilon]\!]\ \delta = \mathbf{poly}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta(A := [\![\upsilon]\!], \mathit{poly}_A := \delta)$ the proposition follows.

- **Case** $V = (T\ \ U) :: \mathfrak{V}$: by induction hypothesis we have

$$\begin{aligned}
&(\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{V}]\!]\varrho, \textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{U}\to\mathfrak{V}} \\
&\quad \equiv \forall \upsilon\ \delta\,.\,(\upsilon, \delta) \in \mathcal{S}^{\mathfrak{U}} \\
&\qquad\qquad \supset ((\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{V}]\!]\varrho)\ \upsilon, \textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle\eta\ [\![\upsilon]\!]\ \delta) \in \mathcal{S}^{\mathfrak{V}}
\end{aligned}$$

and
$$(\mathcal{B}[\![U :: \mathfrak{U}]\!]\varrho, \textsf{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{U}}.$$

Setting $\upsilon = \mathcal{B}[\![U :: \mathfrak{U}]\!]\varrho$ and $\delta = \textsf{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle\eta$ and since $[\![\mathcal{B}[\![U]\!]\varrho]\!] = [\![U]\!]\eta$, we obtain

$$((\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{V}]\!]\varrho)\ (\mathcal{B}[\![U :: \mathfrak{U}]\!]\varrho), (\textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle\eta)\ ([\![U]\!]\eta)\ (\textsf{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle\eta)) \in \mathcal{S}^{\mathfrak{V}}.$$

- **Case** $V = \mathit{Fix}\ T :: \mathfrak{U}$: by induction hypothesis we have

$$\begin{aligned}
&(\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\varrho, \textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta) \in \mathcal{S}^{\mathfrak{U}\to\mathfrak{U}} \\
&\quad \equiv \forall \upsilon\ \delta\,.\,(\upsilon, \delta) \in \mathcal{S}^{\mathfrak{U}} \\
&\qquad\qquad \supset (\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\varrho\ \upsilon, \textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta\ [\![\upsilon]\!]\ \delta) \in \mathcal{S}^{\mathfrak{U}}.
\end{aligned}$$

Define
$$\begin{aligned}
\tau_0 &= \bot \\
\tau_{n+1} &= \mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\varrho\ \tau_n \\
\delta_0 &= \bot \\
\delta_{n+1} &= \textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta\ [\![\tau_n]\!]\ \delta_n.
\end{aligned}$$

Using the induction hypothesis and the fact that $\mathcal{S}^{\mathfrak{V}}$ is pointed (since **poly** is strict) we can show
$$(\tau_n, \delta_n) \in \mathcal{S}^{\mathfrak{U}}$$
for all $n \in \mathbb{N}$. Since $\mathcal{S}^{\mathfrak{V}}$ is furthermore chain-complete, we have
$$(\bigsqcup\{\tau_n \mid n \in \mathbb{N}\}, \bigsqcup\{\delta_n \mid n \in \mathbb{N}\}) \in \mathcal{S}^{\mathfrak{U}}.$$

Now, since $\bigsqcup\{\tau_n \mid n \in \mathbb{N}\} = \textsf{lfp}\ (\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\varrho)$ and

$$\begin{aligned}
&\bigsqcup\{\delta_n \mid n \in \mathbb{N}\} \\
= &\quad \{\ \text{definition } \textsf{slfp}\ \} \\
&\textsf{slfp}\ ([\![\mathcal{B}[\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\varrho]\!])\ (\textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta) \\
= &\quad \{\ [\![\mathcal{B}[\![T]\!]\varrho]\!] = [\![T]\!]\eta\ \} \\
&\textsf{slfp}\ ([\![T :: \mathfrak{U} \to \mathfrak{U}]\!]\eta)\ (\textsf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle\eta)
\end{aligned}$$

the proposition follows. □

Theorem 3.6 Let T be a closed monomorphic type term of kind $\star$, then
$$\textsf{poly}\langle \mathrm{BT}(T) \rangle = \textsf{poly}\langle\!\langle T \rangle\!\rangle.$$ □

3.2 Generalizing to first- and second-order kinds

In the previous section we have considered generic values indexed by types of kind $\star$. For generic values such as *size* that are indexed by type constructors, some additional machinery is needed. Before we tackle the general case, we first discuss the main ideas for type indices of kind $\star \to \star$ (Section 3.2.1) and kind $(\star \to \star) \to \star \to \star$ (Section 3.2.2). Sections 3.2.3–3.2.5 then mirror the structure of the previous section. Section 3.2.3 characterizes the set of normal forms of types of first- or second-order kind, Section 3.2.4 introduces a scheme for defining values indexed by types of this kind, and Section 3.2.5 shows how to promote a generic value thus defined to types of arbitrary kinds. Finally, Section 3.2.6 explains why the approach is limited to types of first- and second-order kinds.

3.2.1 Type indices of kind $\star \to \star$

Generic values such as *size* that are indexed by type constructors of kind $\star \to \star$ are defined using a scheme similar to the one introduced in Section 3.1.2, except that the type patterns on the left-hand side operate on type constructors and that there is one additional case to take into account, namely the identity type. As an example, consider the generic definition of the mapping function (as usual, we use Haskell syntax, that is, we omit universal abstractions and applications).

$$
\begin{array}{lcl}
map\langle T :: \star \to \star\rangle & :: & \forall A\ B\,.\,(A \to B) \to (T\ A \to T\ B) \\
map\langle Id\rangle\ m\ a & = & m\ a \\
map\langle \underline{1}\rangle\ m\ u & = & u \\
map\langle \underline{Char}\rangle\ m\ c & = & c \\
map\langle \underline{Int}\rangle\ m\ i & = & i \\
map\langle F \mathbin{\underline{+}} G\rangle\ m\ (inl\ f) & = & inl\ (map\langle F\rangle\ m\ f) \\
map\langle F \mathbin{\underline{+}} G\rangle\ m\ (inr\ g) & = & inr\ (map\langle G\rangle\ m\ g) \\
map\langle F \mathbin{\underline{\times}} G\rangle\ m\ (f, g) & = & (map\langle F\rangle\ m\ f, map\langle G\rangle\ m\ g)
\end{array}
$$

The definition employs the type abbreviations introduced in Section 2.5.1. We will refer to $\underline{1}$, $\underline{Char}$, $\underline{Int}$, $(\underline{+})$ and $(\underline{\times})$ as the *lifted variants* of 1, $Char$, Int, $(+)$ and $(\times)$, respectively. When used as type patterns, we call *Id projection pattern* and $\underline{1}$, $\underline{Char}$, $\underline{Int}$, $F \mathbin{\underline{+}} G$ and $F \mathbin{\underline{\times}} G$ *constructor patterns.*

The mapping function $map\langle T\rangle$ applies a given function to each element of type A in a given container of type $T\ A$. The above definition shows quite clearly that the mapping function leaves the structure of the container intact. We have remarked several times that the mapping function is related to the categorical concept of a functor—$map\langle T\rangle$ corresponds to the morphism part of a functor the object part being given by the type constructor T. Now, the above definition of *map* makes essential use of the fact that Id, $\underline{1}$, $\underline{Char}$, $\underline{Int}$, $(+)$ and $(\times)$ are functors (or bifunctors) themselves. Using the mapping functions of these type constructors we can define *map* more succinctly:

$$
\begin{array}{lcl}
map\langle T :: \star \to \star\rangle & :: & \forall A\ B\,.\,(A \to B) \to (T\ A \to T\ B) \\
map\langle Id\rangle\ m & = & m \\
map\langle \underline{1}\rangle\ m & = & id \\
map\langle \underline{Char}\rangle\ m & = & id \\
map\langle \underline{Int}\rangle\ m & = & id \\
map\langle F \mathbin{\underline{+}} G\rangle\ m & = & map\langle F\rangle\ m + map\langle G\rangle\ m \\
map\langle F \mathbin{\underline{\times}} G\rangle\ m & = & map\langle F\rangle\ m \times map\langle G\rangle\ m.
\end{array}
$$

Now, can we be sure that the type patterns cover all possible cases? To answer this question let us characterize the set of normal forms of types of kind $\star \to \star$. Assume that we are given a type F of kind $\star \to \star$. Applying η-expansion we have $F = \Lambda A \,.\, F\ A$. The body of the abstraction has kind $\star$ and we know from Section 3.1.1 the shape of its normal form. The free type variable, A, is simply treated as an additional type constant of kind $\star$. Now, to make the passing of A explicit we abstract A out. The abstraction function $[A]\ T$ is defined by induction on the structure of normal forms of kind $\star$.

$$
\begin{array}{lcl}
[A]\,A & = & \mathit{Id} \\
[A]\,1 & = & \underline{1} \\
[A]\,\mathit{Char} & = & \underline{\mathit{Char}} \\
[A]\,\mathit{Int} & = & \underline{\mathit{Int}} \\
[A]\,T + U & = & ([A]\ T)\ \underline{+}\ ([A]\ U) \\
[A]\,T \times U & = & ([A]\ T)\ \underline{\times}\ ([A]\ U) \\
[A]\,T \to U & = & ([A]\ T)\ \underline{\to}\ ([A]\ U)
\end{array}
$$

The abstraction process replaces A by Id and the primitive type constructors by their lifted versions. It is easy to see that $\Lambda A \,.\, T = [A]\ T$. Thus, we obtain the following characterization of $\mathrm{NF}^{\star\to\star}$.

$$
\begin{array}{lcl}
\mathrm{NF}^{\star\to\star} & ::= & \mathit{Id} \\
& | & \underline{1} \\
& | & \underline{\mathit{Char}} \\
& | & \underline{\mathit{Int}} \\
& | & \mathrm{NF}_1^{\star\to\star}\ \underline{+}\ \mathrm{NF}_2^{\star\to\star} \\
& | & \mathrm{NF}_1^{\star\to\star}\ \underline{\times}\ \mathrm{NF}_2^{\star\to\star} \\
& | & \mathrm{NF}_1^{\star\to\star}\ \underline{\to}\ \mathrm{NF}_2^{\star\to\star}
\end{array}
$$

We can, in fact, view Id, $\underline{1}$, $\underline{\mathit{Char}}$, $\underline{\mathit{Int}}$, '$\underline{+}$', '$\underline{\times}$' and '$\underline{\to}$' as a tiny combinator language for defining type constructors of kind $\star \to \star$.

3.2.2 Type indices of kind $(\star \to \star) \to \star \to \star$

Let us start with a characterization of the set of normal forms of types of kind $(\star \to \star) \to \star \to \star$. We proceed exactly as in the previous section. Given a type H of kind $(\star \to \star) \to \star \to \star$ we apply η-expansion to obtain $H = \Lambda A_1\ A_2 \,.\, H\ A_1\ A_2$. The body of the abstraction has kind $\star$ and its normal form can be characterized in the usual way. Again, the type variables $A_1 :: \star \to \star$ and $A_2 :: \star$ are treated as additional type constants. The abstraction function now simultaneously abstracts A_1 and A_2 out (note that A_1 takes an argument since it has kind $\star \to \star$):

$$
\begin{array}{lcl}
[A_1\ A_2]\,(A_1\ T) & = & P_1\ ([A_1\ A_2]\ T) \\
[A_1\ A_2]\,A_2 & = & P_2 \\
[A_1\ A_2]\,1 & = & \underline{1} \\
[A_1\ A_2]\,\mathit{Char} & = & \underline{\mathit{Char}} \\
[A_1\ A_2]\,\mathit{Int} & = & \underline{\mathit{Int}} \\
[A_1\ A_2]\,T + U & = & ([A_1\ A_2]\ T)\ \underline{+}\ ([A_1\ A_2]\ U) \\
[A_1\ A_2]\,T \times U & = & ([A_1\ A_2]\ T)\ \underline{\times}\ ([A_1\ A_2]\ U) \\
[A_1\ A_2]\,T \to U & = & ([A_1\ A_2]\ T)\ \underline{\to}\ ([A_1\ A_2]\ U).
\end{array}
$$

The type combinators are defined as follows:

$$\begin{array}{lcl}
P_1\ H & = & \Lambda A_1\ A_2\,.\,A_1\ (H\ A_1\ A_2) \\
P_2 & = & \Lambda A_1\ A_2\,.\,A_2 \\
\underline{1} & = & \Lambda A_1\ A_2\,.\,1 \\
\underline{\mathit{Char}} & = & \Lambda A_1\ A_2\,.\,\mathit{Char} \\
\underline{\mathit{Int}} & = & \Lambda A_1\ A_2\,.\,\mathit{Int} \\
H_1 \mathbin{\underline{+}} H_2 & = & \Lambda A_1\ A_2\,.\,(H_1\ A_1\ A_2) + (H_2\ A_1\ A_2) \\
H_1 \mathbin{\underline{\times}} H_2 & = & \Lambda A_1\ A_2\,.\,(H_1\ A_1\ A_2) \times (H_2\ A_1\ A_2) \\
H_1 \mathbin{\underline{\rightarrow}} H_2 & = & \Lambda A_1\ A_2\,.\,(H_1\ A_1\ A_2) \rightarrow (H_2\ A_1\ A_2).
\end{array}$$

Since we have two type variables, A_1 and A_2, we have two projection patterns, $P_1\ H$ and P_2. Consequently, the set of normal forms $\mathrm{NF}^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star}$ is characterized by the following grammar.

$$\begin{array}{lcl}
\mathrm{NF}^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} & ::= & P_1\ \mathrm{NF}^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \\
& \mid & P_2 \\
& \mid & \underline{1} \\
& \mid & \underline{\mathit{Char}} \\
& \mid & \underline{\mathit{Int}} \\
& \mid & \mathrm{NF}_1^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \mathbin{\underline{+}} \mathrm{NF}_2^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \\
& \mid & \mathrm{NF}_1^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \mathbin{\underline{\times}} \mathrm{NF}_2^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \\
& \mid & \mathrm{NF}_1^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star} \mathbin{\underline{\rightarrow}} \mathrm{NF}_2^{(\star\rightarrow\star)\rightarrow\star\rightarrow\star}
\end{array}$$

An example of a generic function that is indexed by types of this kind is a so-called *higher-order mapping function*. A higher-order functor operates on a *functor category*, which has as objects functors and as arrows natural transformations. In Haskell we can model natural transformations by polymorphic functions:

$$\textbf{type}\ F_1 \mathbin{\dot{\rightarrow}} F_1 \quad = \quad \forall A\,.\,F_1\ A \rightarrow F_1\ A.$$

A natural transformation between functors F_1 and F_2 is simply a polymorphic function of type $F_1 \mathbin{\dot{\rightarrow}} F_2$. A higher-order functor H then consists of a type constructor of kind $(\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$, such as *GRose*, and an associated mapping function of type $(F_1 \mathbin{\dot{\rightarrow}} F_2) \rightarrow (H\ F_1 \mathbin{\dot{\rightarrow}} H\ F_2)$. The mapping function enjoys the following generic definition.

$$\begin{array}{lcl}
\mathit{hmap}\langle T :: (\star\rightarrow\star)\rightarrow\star\rightarrow\star\rangle & :: & \forall F_1\ F_2\,.\,(\mathit{Functor}\ F_1, \mathit{Functor}\ F_2) \\
& & \qquad \Rightarrow (F_1 \mathbin{\dot{\rightarrow}} F_2) \rightarrow (T\ F_1 \mathbin{\dot{\rightarrow}} T\ F_2) \\
\mathit{hmap}\langle P_1\ H\rangle\ m & = & m \cdot \mathit{fmap}\ (\mathit{hmap}\langle H\rangle\ m) \\
\mathit{hmap}\langle P_2\rangle\ m & = & \mathit{id} \\
\mathit{hmap}\langle \underline{1}\rangle\ m & = & \mathit{id} \\
\mathit{hmap}\langle \underline{\mathit{Char}}\rangle\ m & = & \mathit{id} \\
\mathit{hmap}\langle \underline{\mathit{Int}}\rangle\ m & = & \mathit{id} \\
\mathit{hmap}\langle H_1 \mathbin{\underline{+}} H_2\rangle\ m\ (\mathit{inl}\ h_1) & = & \mathit{inl}\ (\mathit{hmap}\langle H_1\rangle\ m\ h_1) \\
\mathit{hmap}\langle H_1 \mathbin{\underline{+}} H_2\rangle\ m\ (\mathit{inr}\ h_2) & = & \mathit{inr}\ (\mathit{hmap}\langle H_2\rangle\ m\ h_2) \\
\mathit{hmap}\langle H_1 \mathbin{\underline{\times}} H_2\rangle\ m\ (h_1, h_2) & = & (\mathit{hmap}\langle H_1\rangle\ m\ h_1, \mathit{hmap}\langle H_2\rangle\ m\ h_2)
\end{array}$$

Note that the assumption that F_1 and F_2 are functors is expressed by the Haskell context $(\mathit{Functor}\ F_1, \mathit{Functor}\ F_2)$. Actually, we only require a simpler context, namely $\mathit{Functor}\ F_1$, since the method of *Functor*, *fmap*, is only used at that type. There is an alternative definition of *hmap* given by (only the first equation is different)

$$\mathit{hmap}\langle P_1\ H\rangle\ m \quad = \quad \mathit{fmap}\ (\mathit{hmap}\langle H\rangle\ m) \cdot m$$

that requires a *Functor* F_2 context. Both definitions are equivalent by virtue of m's naturality condition (or by virtue of the parametricity theorem).

We can use the higher-order map, for instance, to change the 'base collection' F in a generalized rose tree of type *GRose F A*. Say, we are given a function $toSequ :: List \stackrel{.}{\to} Sequ$ that turns a list into a binary random-access list. Using $hmap\langle GRose\rangle\ toSequ :: GRose\ List \stackrel{.}{\to} GRose\ Sequ$ we can change the base collection of a generalized rose tree from lists to binary random-access lists. Note that the higher-order map does not touch the elements contained in the tree. The elements can be changed using $map\langle GRose\ List\rangle$ or $map\langle GRose\ Sequ\rangle$.

3.2.3 Normal forms of types

After having considered two instances, let us tackle the general case. To this end assume that we are given an arbitrary set of type constants $Const = \{ C_1 :: \mathfrak{C}_1, \ldots, C_l :: \mathfrak{C}_l \}$ where the kind of the i-th constant C_i is given by $\mathfrak{C}_i = \star^{k_i} \to \star$. So the only requirement on *Const* is that the type constants have first-order kinds (note that if $order(\mathfrak{C}) \leqslant 1$ then $\mathfrak{C} = \star^k \to \star$ for some k). Furthermore, assume that we want to define a generic value that is indexed by a type constructor of kind $\mathfrak{P} = \mathfrak{P}_1 \to \cdots \to \mathfrak{P}_n \to \star$ with $\mathfrak{P}_j = \star^{m_j} \to \star$. So $\mathfrak{P}$ has at most order 2.

For characterizing the set of normal forms it is useful to introduce the notion of *lifting*. We have already introduced lifting in Section 2.4.4 albeit for terms of the simply typed λ-calculus. The following is a recap of the definitions adapted to type terms. Lifting maps a type $T :: \mathfrak{T}$ to a type $\uparrow T :: \uparrow\mathfrak{T}$ where $\uparrow\mathfrak{T}$ is given by

$$\begin{array}{lcl} \uparrow\star & = & \mathfrak{P} \\ \uparrow\mathfrak{T} \times \mathfrak{U} & = & (\uparrow\mathfrak{T}) \times (\uparrow\mathfrak{U}) \\ \uparrow\mathfrak{T} \to \mathfrak{U} & = & (\uparrow\mathfrak{T}) \to (\uparrow\mathfrak{U}). \end{array}$$

Note that $\uparrow\mathfrak{T}$ is defined by induction on the structure of kind terms. If $\mathfrak{P} = \star$, then $(\uparrow)$ is simply the identity. The lifted version $\uparrow T$ of type T is defined by induction on the structure of type terms:

$$\begin{array}{lcl} \uparrow C & = & \underline{C} \\ \uparrow A & = & \underline{A} \\ \uparrow(T_1, T_2) & = & (\uparrow T_1, \uparrow T_2) \\ \uparrow Outl\ T & = & Outl\ (\uparrow T) \\ \uparrow Outr\ T & = & Outr\ (\uparrow T) \\ \uparrow\Lambda A\,.\,T & = & \Lambda\underline{A}\,.\uparrow T \\ \uparrow T\ U & = & (\uparrow T)\ (\uparrow U) \\ \uparrow Fix\ T & = & Fix\ (\uparrow T). \end{array}$$

The transformation assumes that for each type variable $A :: \mathfrak{A}$ there is a lifted type variable $\underline{A} :: \uparrow\mathfrak{A}$. The lifted versions of the primitive types $\underline{C_i} :: \uparrow\mathfrak{C}_i$ are given by

$$\begin{array}{l} \underline{C_i}\ (H_1 :: \mathfrak{P})\ \ldots\ (H_{k_i} :: \mathfrak{P}) \\ \quad = \Lambda(X_1 :: \mathfrak{P}_1)\ \ldots\ (X_n :: \mathfrak{P}_n)\,.\,C_i\ (H_1\ X_1\ \ldots\ X_n)\ \ldots\ (H_{k_i}\ X_1\ \ldots\ X_n). \end{array}$$

Recall from the previous two sections that the type patterns used in a generic definition can be divided into two categories: *projection patterns* and *constructor patterns*. The latter are formed using the lifted type constants. The former are built using the projection types $P_j :: \uparrow\mathfrak{P}_j$ defined by

$$\begin{array}{l} P_j\ (H_1 :: \mathfrak{P})\ \ldots\ (H_{m_j} :: \mathfrak{P}) \\ \quad = \Lambda(X_1 :: \mathfrak{P}_1)\ \ldots\ (X_n :: \mathfrak{P}_n)\,.\,X_j\ (H_1\ X_1\ \ldots\ X_n)\ \ldots\ (H_{m_j}\ X_1\ \ldots\ X_n). \end{array}$$

Consequently, the set of normal forms is characterized by the following grammar.

$$\begin{array}{rcl} \mathrm{NF}^{\mathfrak{P}} & ::= & P_1\ \mathrm{NF}_1^{\mathfrak{P}}\ \ldots\ \mathrm{NF}_{m_1}^{\mathfrak{P}} \\ & | & \ldots \\ & | & P_n\ \mathrm{NF}_1^{\mathfrak{P}}\ \ldots\ \mathrm{NF}_{m_n}^{\mathfrak{P}} \\ & | & \underline{C_1}\ \mathrm{NF}_1^{\mathfrak{P}}\ \ldots\ \mathrm{NF}_{k_1}^{\mathfrak{P}} \\ & | & \ldots \\ & | & \underline{C_l}\ \mathrm{NF}_1^{\mathfrak{P}}\ \ldots\ \mathrm{NF}_{k_l}^{\mathfrak{P}} \end{array}$$

We have n projection patterns and l constructor patterns. Thus, the total number of patterns depends on the arity of $\mathfrak{P}$ and on the number of primitive types.

The following lemma, which will be needed in Section 3.2.5, shows that if we apply a lifted type to the projection types we obtain the original type back.

LEMMA 3.7 Let $T :: \mathfrak{P}$ be a closed monomorphic type term, then

$$(\uparrow T)\ P_1\ \ldots\ P_n \quad \approx \quad T.$$

PROOF. The proof is based on Corollary 2.22 (we use the Böhm tree model as the underlying model $\mathcal{E}$). First of all, we require type-level counterparts of the combinators K and S.

$$\begin{array}{lcl} \hat{K}\ T & = & \Lambda(X_1 :: \mathfrak{P}_1)\ \ldots\ (X_n :: \mathfrak{P}_n) \,.\, T \\ \hat{S}\ T\ U & = & \Lambda(X_1 :: \mathfrak{P}_1)\ \ldots\ (X_n :: \mathfrak{P}_n) \,.\, (T\ X_1\ \ldots\ X_n)\ (U\ X_1\ \ldots\ X_n) \end{array}$$

Let $\hat{\mathsf{S}} = \mathcal{B}[\![\hat{S}]\!]$ and note that $\mathsf{app}^{\mathcal{K}}\ \varphi\ d = \hat{\mathsf{S}}\ \varphi\ d$. Now, Corollary 2.22 states that $\mathcal{B}[\![\hat{K}\ T]\!] \sim^{\mathfrak{P}} \mathcal{B}[\![\uparrow T]\!]$, which is equivalent to

$$\begin{array}{l} \forall \tau_1\ \ldots\ \tau_n \,.\, \forall \tau_1'\ \ldots\ \tau_n' \,. \\ \quad \tau_1 \sim^{\mathfrak{P}_1} \tau_1' \cap \cdots \cap \tau_n \sim^{\mathfrak{P}_n} \tau_n' \\ \qquad \supset \hat{\mathsf{S}}\ \cdots\ (\hat{\mathsf{S}}\ (\mathcal{B}[\![\hat{K}\ T]\!])\ \tau_1)\ \cdots\ \tau_n = \mathcal{B}[\![\uparrow T]\!]\ \tau_1'\ \cdots\ \tau_n', \end{array} \tag{3.1}$$

where

$$\tau_j \sim^{\mathfrak{P}_j} \tau_j' \quad \equiv \quad \forall \upsilon_1\ \ldots\ \upsilon_{m_j} \,.\, \hat{\mathsf{S}}\ \cdots\ (\hat{\mathsf{S}}\ \tau_j\ \upsilon_1)\ \cdots\ \upsilon_{m_j} = \tau_j'\ \upsilon_1\ \cdots\ \upsilon_{m_j}.$$

Note that $\mathcal{B}[\![Nth_j]\!] \sim^{\mathfrak{P}_j} \mathcal{B}[\![P_j]\!]$ where $Nth_j = \Lambda(X_1 :: \mathfrak{P}_1)\ \ldots\ (X_n :: \mathfrak{P}_n) \,.\, X_j$, which implies

$$\begin{array}{cl} & \mathcal{B}[\![(\uparrow T)\ P_1\ \ldots\ P_n]\!] \\ = & \quad \{ \text{ definition of } \mathcal{B} \ \} \\ & \mathcal{B}[\![\uparrow T]\!]\ \mathcal{B}[\![P_1]\!]\ \cdots\ \mathcal{B}[\![P_n]\!] \\ = & \quad \{ \text{ (3.1) and } \mathcal{B}[\![Nth_j]\!] \sim^{\mathfrak{P}_j} \mathcal{B}[\![P_j]\!] \ \} \\ & \hat{\mathsf{S}}\ \ldots\ (\hat{\mathsf{S}}\ (\mathcal{B}[\![\hat{K}\ T]\!])\ \mathcal{B}[\![Nth_1]\!])\ \cdots\ \mathcal{B}[\![Nth_n]\!] \\ = & \quad \{ \text{ definition of } \mathcal{B} \ \} \\ & \mathcal{B}[\![\hat{S}\ \ldots\ (\hat{S}\ (\hat{K}\ T)\ Nth_1)\ \ldots\ Nth_n]\!] \\ = & \quad \{ \text{ definition of } \hat{S},\ \hat{K} \text{ and } Nth_j \ \} \\ & \mathcal{B}[\![\Lambda X_1\ \ldots\ X_n \,.\, T\ X_1\ \ldots\ X_n]\!] \\ = & \quad \{ \ \eta\text{-conversion } \} \\ & \mathcal{B}[\![T]\!]. \end{array}$$

Consequently, $(\uparrow T)\ P_1\ \ldots\ P_n \approx T$. □

For instance, if $\mathfrak{P} = \star \to \star$, we have $P_1 = \mathit{Id}$ and $(\uparrow T)\ \mathit{Id} = T$. As a second example, if $\mathfrak{P} = \star \to \star \to \star$, we have $P_1 = \mathit{Fst} = \Lambda A_1\ A_2 \,.\, A_1$, $P_2 = \mathit{Snd} = \Lambda A_1\ A_2 \,.\, A_2$ and $(\uparrow T)\ \mathit{Fst}\ \mathit{Snd} = T$.

3.2.4 Defining generic values

The characterization of normal forms given in the previous section suggests the following scheme for defining values indexed by type constructors of kind $\mathfrak{P}$.

$$
\begin{array}{lcl}
\mathit{poly}\langle T :: \mathfrak{P}\rangle & :: & \mathit{Poly}\ T \\
\mathit{poly}\langle P_1\ A_1\ \ldots\ A_{m_1}\rangle & = & \mathit{poly}_{P_1}\ A_1\ (\mathit{poly}\langle A_1\rangle)\ \ldots\ A_{m_1}\ (\mathit{poly}\langle A_{m_1}\rangle) \\
\ldots & & \\
\mathit{poly}\langle P_n\ A_1\ \ldots\ A_{m_n}\rangle & = & \mathit{poly}_{P_n}\ A_1\ (\mathit{poly}\langle A_1\rangle)\ \ldots\ A_{m_n}\ (\mathit{poly}\langle A_{m_n}\rangle) \\
\mathit{poly}\langle \underline{C_1}\ A_1\ \ldots\ A_{k_1}\rangle & = & \mathit{poly}_{\underline{C_1}}\ A_1\ (\mathit{poly}\langle A_1\rangle)\ \ldots\ A_{k_1}\ (\mathit{poly}\langle A_{k_1}\rangle) \\
\ldots & & \\
\mathit{poly}\langle \underline{C_l}\ A_1\ \ldots\ A_{k_l}\rangle & = & \mathit{poly}_{\underline{C_l}}\ A_1\ (\mathit{poly}\langle A_1\rangle)\ \ldots\ A_{k_l}\ (\mathit{poly}\langle A_{k_l}\rangle)
\end{array}
$$

The type of $\mathit{poly}\langle T\rangle$ is given by $\mathit{Poly}\ T$, where Poly is a type constructor of kind $\mathfrak{P} \to \star$. The poly_{P_j} and the $\mathit{poly}_{\underline{C_i}}$ values must have the following types:

$$
\begin{array}{lcl}
\mathit{poly}_{P_j} & :: & \forall A_1 \,.\, \mathit{Poly}\ A_1 \to \cdots \to \forall A_{m_j} \,.\, \mathit{Poly}\ A_{m_j} \to \mathit{Poly}\ (P_j\ A_1\ \ldots\ A_{m_j}) \\
\mathit{poly}_{\underline{C_i}} & :: & \forall A_1 \,.\, \mathit{Poly}\ A_1 \to \cdots \to \forall A_{k_i} \,.\, \mathit{Poly}\ A_{k_i} \to \mathit{Poly}\ (\underline{C_i}\ A_1\ \ldots\ A_{k_i}).
\end{array}
$$

Each of the generic definitions we have encountered so far adheres to this definitional scheme. As an example, let us consider how the *size* function introduced in the introduction fits into it: *size* is indexed by type constructors of kind $\star \to \star$, $\mathit{size}\langle T\rangle$ has type $\mathit{Size}\ T = \forall A \,.\, T\ A \to \mathit{Int}$ and the functions $\mathit{size}_{\mathit{Id}}$, $\mathit{size}_{\underline{1}}$, $\mathit{size}_{\underline{\mathit{Int}}}$, $\mathit{size}_{\underline{\mathit{Char}}}$, $\mathit{size}_{\underline{+}}$, and $\mathit{size}_{\underline{\times}}$ are given by

$$
\begin{array}{lcl}
\mathit{size}_{\mathit{Id}} & = & \lambda A \,.\, \lambda a :: A \,.\, 1 \\
\mathit{size}_{\underline{1}} & = & \lambda A \,.\, \lambda u :: 1 \,.\, 0 \\
\mathit{size}_{\underline{\mathit{Char}}} & = & \lambda A \,.\, \lambda c :: \mathit{Char} \,.\, 0 \\
\mathit{size}_{\underline{\mathit{Int}}} & = & \lambda A \,.\, \lambda i :: \mathit{Int} \,.\, 0 \\
\mathit{size}_{\underline{+}} & = & \lambda F \,.\, \lambda \mathit{size}_F :: (\forall A \,.\, F\ A \to \mathit{Int}) \,.\, \lambda G \,.\, \lambda \mathit{size}_G :: (\forall A \,.\, G\ A \to \mathit{Int}) \,. \\
 & & \lambda A \,.\, \lambda s :: (F\ A + G\ A) \,.\, \mathbf{case}\ s\ \mathbf{of}\ \{\mathit{inl}\ f \Rightarrow \mathit{size}_F\ A\ f; \mathit{inr}\ g \Rightarrow \mathit{size}_G\ A\ g\} \\
\mathit{size}_{\underline{\times}} & = & \lambda F \,.\, \lambda \mathit{size}_F :: (\forall A \,.\, F\ A \to \mathit{Int}) \,.\, \lambda G \,.\, \lambda \mathit{size}_G :: (\forall A \,.\, G\ A \to \mathit{Int}) \,. \\
 & & \lambda A \,.\, \lambda p :: (F\ A \times G\ A) \,.\, \mathit{size}_F\ A\ (\mathit{outl}\ p) + \mathit{size}_G\ A\ (\mathit{outr}\ p).
\end{array}
$$

As an aside, note that $\mathit{size}\langle T\rangle$ is not only a generic, but also a polymorphic function. This combination is, however, not compelling: the generic function $\mathit{sum}\langle T\rangle$, which sums a structure of integers, has the monomorphic type $T\ \mathit{Int} \to \mathit{Int}$.

The semantics of generic definitions is as before: the meaning of $\mathit{poly}\langle T\rangle$, where $T \in \mathit{MonoType}$ is a closed monomorphic type term of kind $\mathfrak{P}$, is given by $\mathsf{\mathbf{poly}}\langle \mathrm{BT}(T)\rangle$.

3.2.5 Specializing generic values

Promoting *poly* to types of arbitrary kind also proceeds as before, except that we are now working in a higher realm, that is, we work with lifted kinds and types.

To begin with, the type of the promoted version is given by $\mathit{Poly}\langle - \rangle$, which is defined by induction on the structure of lifted kinds.

$$\begin{array}{lcl}
\mathit{Poly}\langle \uparrow\mathfrak{T} :: \Box \rangle & :: & (\uparrow\mathfrak{T}) \to \star \\
\mathit{Poly}\langle \uparrow\star \rangle\ T & = & \mathit{Poly}\ T \\
\mathit{Poly}\langle \uparrow\mathfrak{A} \times \mathfrak{B} \rangle\ T & = & \mathit{Poly}\langle \uparrow\mathfrak{A} \rangle\ (\mathit{Outl}\ T) \times \mathit{Poly}\langle \uparrow\mathfrak{B} \rangle\ (\mathit{Outr}\ T) \\
\mathit{Poly}\langle \uparrow\mathfrak{A} \to \mathfrak{B} \rangle\ T & = & \forall A\,.\,\mathit{Poly}\langle \uparrow\mathfrak{A} \rangle\ A \to \mathit{Poly}\langle \uparrow\mathfrak{B} \rangle\ (T\ A)
\end{array}$$

The definition of $\mathit{poly}\langle\!\langle - \rangle\!\rangle$ is inductive on the structure of kinding derivations.

$$\begin{array}{lcl}
\mathit{poly}\langle\!\langle \uparrow T :: \uparrow\mathfrak{T} \rangle\!\rangle & :: & \mathit{Poly}\langle \uparrow\mathfrak{T} \rangle\ (\uparrow T) \\
\mathit{poly}\langle\!\langle \underline{C} :: \mathfrak{C} \rangle\!\rangle & = & \mathit{poly}_{\underline{C}} \\
\mathit{poly}\langle\!\langle \underline{A} :: \mathfrak{A} \rangle\!\rangle & = & \mathit{poly}_{\underline{A}} \\
\mathit{poly}\langle\!\langle (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle & = & (\mathit{poly}\langle\!\langle T_1 :: \mathfrak{T}_1 \rangle\!\rangle, \mathit{poly}\langle\!\langle T_2 :: \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Outl}\ T :: \mathfrak{T}_1 \rangle\!\rangle & = & \mathit{outl}\ (\mathit{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Outr}\ T :: \mathfrak{T}_2 \rangle\!\rangle & = & \mathit{outr}\ (\mathit{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle (\Lambda \underline{A}\,.\,T) :: (\mathfrak{S} \to \mathfrak{T}) \rangle\!\rangle & = & \lambda \underline{A}\,.\,\lambda \mathit{poly}_{\underline{A}}\,.\,\mathit{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle \\
\mathit{poly}\langle\!\langle T\ U :: \mathfrak{V} \rangle\!\rangle & = & (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle)\ U\ (\mathit{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Fix}\ T :: \mathfrak{U} \rangle\!\rangle & = & \mathit{fix}\ ((\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle)\ (\mathit{Fix}\ T)).
\end{array}$$

Note that $\mathit{poly}\langle\!\langle - \rangle\!\rangle$ depends only on the $\mathit{poly}_{\underline{C_i}}$ but not on the poly_{P_j} values. The latter instances are used in the initial call: the specialized version of $\mathit{poly}\langle T \rangle$, which we write poly_T, is given by

$$\mathit{poly}_T \quad = \quad \mathit{poly}\langle\!\langle \uparrow T \rangle\!\rangle\ P_1\ (\mathit{poly}\langle\!\langle P_1 \rangle\!\rangle)\ \ldots\ P_n\ (\mathit{poly}\langle\!\langle P_n \rangle\!\rangle).$$

Thus, in order to specialize $\mathit{poly}\langle T \rangle$ we specialize $\mathit{poly}\langle\!\langle \uparrow T \rangle\!\rangle$. The resulting function has type

$$\forall X_1\,.\,\mathit{Poly}\langle \uparrow\mathfrak{P}_1 \rangle\ X_1 \to \cdots \to \forall X_n\,.\,\mathit{Poly}\langle \uparrow\mathfrak{P}_n \rangle\ X_n \to \mathit{Poly}\ ((\uparrow T)\ X_1\ \ldots\ X_n).$$

Supplying P_j as type and $\mathit{poly}_{P_j} :: \mathit{Poly}\langle \uparrow\mathfrak{P}_j \rangle\ P_j$ as value arguments we obtain a value of type $\mathit{Poly}\ ((\uparrow T)\ P_1\ \ldots\ P_n) \approx \mathit{Poly}\ T$.

The following theorem states that the specialization is correct.

THEOREM 3.8 Let $T :: \mathfrak{P}$ be a closed monomorphic type term, then

$$\mathbf{poly}\langle \mathrm{BT}(T) \rangle \quad = \quad \mathbf{poly}\langle\!\langle \uparrow T \rangle\!\rangle\ [\![P_1]\!]\ (\mathbf{poly}\langle\!\langle P_1 \rangle\!\rangle)\ \ldots\ [\![P_n]\!]\ (\mathbf{poly}\langle\!\langle P_n \rangle\!\rangle).$$

PROOF. Using an argument similar to Lemma 3.5 we have

$$\begin{array}{l}
(\mathcal{B}[\![\uparrow T]\!], \mathbf{poly}\langle\!\langle \uparrow T \rangle\!\rangle) \in \mathcal{S}^{\mathfrak{P}} \\
\equiv \forall \tau_1\ \ldots\ \tau_n\,.\,\forall \varphi_1\ \ldots\ \varphi_n\,. \\
\qquad (\tau_1, \varphi_1) \in \mathcal{S}^{\mathfrak{P}_1}\ \cap\ \cdots\ \cap\ (\tau_n, \varphi_n) \in \mathcal{S}^{\mathfrak{P}_n} \\
\qquad\quad \supset\ \mathbf{poly}\langle \mathcal{B}[\![\uparrow T]\!]\ \tau_1\ \cdots\ \tau_n \rangle = \mathbf{poly}\langle\!\langle \uparrow T \rangle\!\rangle\ [\![\tau_1]\!]\ \varphi_1\ \ldots\ [\![\tau_n]\!]\ \varphi_n,
\end{array} \qquad (3.2)$$

where

$$\begin{array}{l}
(\tau_j, \varphi_j) \in \mathcal{S}^{\mathfrak{P}_j} \\
\quad \equiv \forall \upsilon_1\ \ldots\ \upsilon_{m_j}\,. \\
\qquad\qquad \mathbf{poly}\langle \tau_j\ \upsilon_1\ \cdots\ \upsilon_{m_j} \rangle = \varphi_j\ [\![\upsilon_1]\!]\ (\mathbf{poly}\langle \upsilon_1 \rangle)\ \ldots\ [\![\upsilon_{m_j}]\!]\ (\mathbf{poly}\langle \upsilon_{m_j} \rangle).
\end{array}$$

Note that $(\mathcal{B}[\![P_j]\!], \textsf{\textbf{poly}}\langle\!\langle P_j \rangle\!\rangle) \in \mathcal{S}^{\mathfrak{P}_j}$, which implies

$$
\begin{array}{ll}
 & \textsf{\textbf{poly}}\langle\!\langle \uparrow T \rangle\!\rangle\ [\![P_1]\!]\ (\textsf{\textbf{poly}}\langle\!\langle P_1 \rangle\!\rangle)\ \ldots\ [\![P_n]\!]\ (\textsf{\textbf{poly}}\langle\!\langle P_n \rangle\!\rangle) \\
= & \quad \{\ (3.2) \text{ and } (\mathcal{B}[\![P_j]\!], \textsf{\textbf{poly}}\langle\!\langle P_j \rangle\!\rangle) \in \mathcal{S}^{\mathfrak{P}_j}\ \} \\
 & \textsf{\textbf{poly}}\langle \mathcal{B}[\![\uparrow T]\!]\ \mathcal{B}[\![P_1]\!]\ \cdots\ \mathcal{B}[\![P_n]\!] \rangle \\
= & \quad \{\ \text{definition of } \mathcal{B}\ \} \\
 & \textsf{\textbf{poly}}\langle \mathcal{B}[\![(\uparrow T)\ P_1\ \cdots\ P_n]\!] \rangle \\
= & \quad \{\ \text{Lemma 3.7}\ \} \\
 & \textsf{\textbf{poly}}\langle \mathcal{B}[\![T]\!] \rangle
\end{array}
\qquad \square
$$

3.2.6 Limitations of the approach

The approach to generic programming introduced in the previous sections is restricted to type constants of first-order kind and type indices of second-order kind.

To see why *Const* must not contain types of second-order kind or higher assume that $\mathit{Fix} :: (\star \to \star) \to \star$ is a primitive type. Since *Fix*'s argument is a type constructor, we can no longer define generic values inductively: $\mathit{poly}\langle \mathit{Fix}\ F \rangle$, for instance, cannot fall back on $\mathit{poly}\langle F \rangle$ since F has not kind $\star$. A similar argument applies to type indices. Recall from the characterization of normal forms that we η-expand a type T of kind $\mathfrak{P}$ to $\Lambda A_1\ \ldots\ A_n \,.\, T\ A_1\ \ldots\ A_n$. The type parameters $A_1, \ldots, A_n$ are then treated like additional type constants. Consequently, their kinds must have order less or equal 1, which in turn implies that $\mathfrak{P}$ must have order less or equal 2.

3.3 Type-indexed values with kind-indexed types

In the two previous sections we have discussed POPL-style generic definitions. They nicely illustrate the power of genericity: to define a generic value for all possible instances of data types it suffices to provide instances for all primitive types (plus some instances for projection types). We have also come across some limitations of the approach: primitive types are restricted to first-order kinded types and type indices may only range over second-order kinded types. One may argue that this is not a severe restriction as higher-order kinded types are a rare species. However, there is one further limitation that is not so obvious at first sight but that is more constraining in practice: type indices are restricted to types of one *fixed* kind.

To illustrate the problem consider again the mapping function. In Section 3.2.1 we have defined a mapping function for unary type constructors of kind $\star \to \star$. But mapping functions can be defined for type constructors of arbitrary arity. In the general case, the mapping function takes n functions and applies the i-th function to each element of type A_i in a given structure of type $F\ A_1\ \ldots\ A_n$. Alas, POPL-style definitions do not allow to define these mapping functions at one stroke. The reason is simply that the mapping functions have different types for different arities. For instance, here is the mapping function for bifunctors:

$$
\begin{array}{lcl}
bimap\langle T :: \star \to \star \to \star \rangle & :: & \forall A_1\ A_2 \, . \, (A_1 \to A_2) \\
 & & \to \forall B_1\ B_2 \, . \, (B_1 \to B_2) \to (T\ A_1\ B_1 \to T\ A_2\ B_2) \\
bimap\langle Fst \rangle\ mA\ mB\ a & = & mA\ a \\
bimap\langle Snd \rangle\ mA\ mB\ b & = & mB\ b \\
bimap\langle \underline{1} \rangle\ mA\ mB\ u & = & u \\
bimap\langle \underline{Char} \rangle\ mA\ mB\ c & = & c \\
bimap\langle \underline{Int} \rangle\ mA\ mB\ i & = & i \\
bimap\langle F \mathbin{\underline{+}} G \rangle\ mA\ mB\ (inl\ f) & = & inl\ (bimap\langle F \rangle\ mA\ mB\ f) \\
bimap\langle F \mathbin{\underline{+}} G \rangle\ mA\ mB\ (inr\ g) & = & inr\ (bimap\langle G \rangle\ mA\ mB\ g) \\
bimap\langle F \mathbin{\underline{\times}} G \rangle\ mA\ mB\ (f, g) & = & (bimap\langle F \rangle\ mA\ mB\ f, bimap\langle G \rangle\ mA\ mB\ g).
\end{array}
$$

The definition is nearly identical to the definition of *map* except for the first two cases. The mapping function for ternary functors also requires a separate definition and it also shares most of the code with *map* and so forth. Somewhat ironically, even though the generic programmer has to provide separate definitions for each arity, the specialization of *map* and colleagues works for arbitrary kinds. If a unary type constructor is defined in terms of, say, a ternary type constructor, then the specialization generates a (higher-order) mapping function for this type.

So the million-dollar question is, whether there is a chance that the generic programmer may profit from the flexibility present at the implementation level. Fortunately, the answer to this question is in the affirmative. But before we spell out the details, let us make a brief détour.

What is the most uninteresting generic function you can think of? Most readers would probably agree that this is the generic identity function. Here is its definition—we call it *copy* because it copies the whole of its argument.

$$
\begin{array}{lcl}
copy\langle T :: \star \rangle & :: & T \to T \\
copy\langle 1 \rangle\ u & = & u \\
copy\langle Char \rangle\ c & = & c \\
copy\langle Int \rangle\ i & = & i \\
copy\langle A + B \rangle\ (inl\ a) & = & inl\ (copy\langle A \rangle\ a) \\
copy\langle A + B \rangle\ (inr\ b) & = & inr\ (copy\langle B \rangle\ b) \\
copy\langle A \times B \rangle\ (a, b) & = & (copy\langle A \rangle\ a, copy\langle B \rangle\ b)
\end{array}
$$

For the sake of example let us specialize the *copy* function to some data types. Recall that the promoted version has type $copy\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle :: Copy\langle \mathfrak{T} \rangle\ T$ where *Copy* is defined by induction on the structure of kinds:

$$
\begin{array}{lcl}
Copy\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \star \\
Copy\langle \star \rangle\ T & = & T \to T \\
Copy\langle \mathfrak{A} \times \mathfrak{B} \rangle\ T & = & Copy\langle \mathfrak{A} \rangle\ (Outl\ T) \times Copy\langle \mathfrak{B} \rangle\ (Outr\ T) \\
Copy\langle \mathfrak{A} \to \mathfrak{B} \rangle\ T & = & \forall A \, . \, Copy\langle \mathfrak{A} \rangle\ A \to Copy\langle \mathfrak{B} \rangle\ (T\ A).
\end{array}
$$

The specialization of *copy* to $List = \Lambda A \, . \, Fix\ (\Lambda B \, . \, 1 + A \times B)$ is, for instance, given by (to improve readability we omit universal abstractions and applications):

$$
\begin{array}{lcl}
copy_{List} & :: & \forall A \, . \, (A \to A) \to (List\ A \to List\ A) \\
copy_{List} & = & \lambda copy_A \, . \, fix\ (\lambda copy_B \, . \, copy_+\ copy_1\ (copy_\times\ copy_A\ copy_B)).
\end{array}
$$

If we rewrite this definition as a Haskell function, we obtain

$$
\begin{array}{lcl}
copyList & :: & \forall A \, . \, (A \to A) \to (List\ A \to List\ A) \\
copyList\ copyA\ nil & = & nil \\
copyList\ copyA\ (cons\ a\ x) & = & cons\ (copyA\ a)\ (copyList\ copyA\ x).
\end{array}
$$

Perhaps surprisingly, the code is identical to *mapList*, the mapping function of *List*. Only the type of *copyList* is more restricted: it takes as first argument a function of type $A \to A$ whereas *mapList* takes a function of type $A_1 \to A_2$. Is this correspondence just a coincidence? Let us take a look at a second example. Specializing *copy* to binary random access lists, $Fork = \Lambda A \, . \, A \times A$ and $Sequ = Fix\ (\Lambda S \, . \, \Lambda A \, . \, 1 + S\ (Fork\ A) + A \times S\ (Fork\ A))$, yields

$$\begin{array}{lcl}
copy_{Fork} & :: & \forall A \, . \, (A \to A) \to (Fork\ A \to Fork\ A) \\
copy_{Fork} & = & \lambda copy_A \, . \, copy_\times\ copy_A\ copy_A \\
copy_{Sequ} & :: & \forall A \, . \, (A \to A) \to (Sequ\ A \to Sequ\ A) \\
copy_{Sequ} & = & fix\ (\lambda copy_S \, . \, \lambda copy_A \, . \, copy_+\ copy_1\ (\\
& & \qquad copy_+\ (copy_S\ (copy_{Fork}\ copy_A)) \\
& & \qquad\quad (copy_\times\ copy_A\ (copy_S\ (copy_{Fork}\ copy_A)))\ )).
\end{array}$$

The corresponding Haskell code looks familiar, as well.

$$\begin{array}{lcl}
copyFork & :: & \forall A \, . \, (A \to A) \to (Fork\ A \to Fork\ A) \\
copyFork\ copyA\ (fork\ a_1\ a_2) & = & fork\ (copyA\ a_1)\ (copyA\ a_2) \\
copySequ & :: & \forall A \, . \, (A \to A) \to (Sequ\ A \to Sequ\ A) \\
copySequ\ copyA\ endS & = & endS \\
copySequ\ copyA\ (zeroS\ s) & = & zeroS\ (copySequ\ (copyFork\ copyA)\ s) \\
copySequ\ copyA\ (oneS\ a\ s) & = & oneS\ (copyA\ a)\ (copySequ\ (copyFork\ copyA)\ s)
\end{array}$$

Again, we obtain the code of the mapping functions!

A first résumé: while the *copy* function is uninteresting and useless when specialized to types of kind $\star$, it is interesting and useful when specialized to type constructors of kind $\star \to \star$ or higher. So why not allow the user to specialize a generic value to types of arbitrary kinds?

Returning to the example one small mismatch remains: the mapping functions have more general types than the instances of *copy*. Can we suitably generalize the type of *copy*? It turns out that we must merely add a second type argument:

$$\begin{array}{lcl}
Map\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \mathfrak{T} \to \star \\
Map\langle \star \rangle\ T_1\ T_2 & = & T_1 \to T_2 \\
Map\langle \mathfrak{T} \times \mathfrak{U} \rangle\ T_1\ T_2 & = & Map\langle \mathfrak{T} \rangle\ (Outl\ T_1)\ (Outl\ T_2) \times Map\langle \mathfrak{U} \rangle\ (Outr\ T_1)\ (Outr\ T_2) \\
Map\langle \mathfrak{T} \to \mathfrak{U} \rangle\ T_1\ T_2 & = & \forall A_1\ A_2 \, . \, Map\langle \mathfrak{T} \rangle\ A_1\ A_2 \to Map\langle \mathfrak{U} \rangle\ (T_1\ A_1)\ (T_2\ A_2).
\end{array}$$

The type of $map\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle$ (alias $copy\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle$) is then $Map\langle \mathfrak{T} \rangle\ T\ T$. It is instructive to consider some instances of *Map*.

$$\begin{array}{lcl}
Map\langle \star \rangle\ Int\ Int & = & Int \to Int \\
Map\langle \star \to \star \rangle\ List\ List & = & \forall A_1\ A_2 \, . \, (A_1 \to A_2) \to (List\ A_1 \to List\ A_2) \\
\multicolumn{3}{l}{Map\langle (\star \to \star) \to \star \to \star \rangle\ GRose\ GRose} \\
& = & \forall F_1\ F_2 \, . \, (\forall B_1\ B_2 \, . \, (B_1 \to B_2) \to (F_1\ B_1 \to F_2\ B_2)) \\
& & \quad \to (\forall A_1\ A_2 \, . \, (A_1 \to A_2) \to (GRose\ F_1\ A_1 \to GRose\ F_2\ A_2))
\end{array}$$

For types of kind $\star$ we obtain the type of the identity function (in fact, *map is* the identity function for types of kind $\star$), for type constructors of kind $\star \to \star$ we obtain the familiar type of mapping functions and for type constructors of kind $(\star \to \star) \to \star \to \star$ we obtain a sort of higher-order map. Note that the first argument of the higher-order map takes a function of type $B_1 \to B_2$ to a function of type $F_1\ B_1 \to F_2\ B_2$, that is, it changes both the container type and the element type. By contrast, the mapping function for *List* (which also has kind $\star \to \star$) takes $A_1 \to A_2$ to $List\ A_1 \to List\ A_2$.

Finally, here is the definition of the mapping function itself. To emphasize that *map* can be specialized to types of arbitrary kinds we enclose the type argument in double angle brackets.

$$
\begin{array}{lcl}
map\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & Map\langle\mathfrak{T}\rangle\ T\ T \\
map\langle\!\langle 1\rangle\!\rangle\ u & = & u \\
map\langle\!\langle Char\rangle\!\rangle\ c & = & c \\
map\langle\!\langle Int\rangle\!\rangle\ i & = & i \\
map\langle\!\langle +\rangle\!\rangle\ mapA\ mapB\ (inl\ a) & = & inl\ (mapA\ a) \\
map\langle\!\langle +\rangle\!\rangle\ mapA\ mapB\ (inr\ b) & = & inr\ (mapB\ b) \\
map\langle\!\langle \times\rangle\!\rangle\ mapA\ mapB\ (a, b) & = & (mapA\ a, mapB\ b)
\end{array}
$$

This straightforward definition contains all the ingredients needed to derive *map*s for arbitrary data types of arbitrary kinds. We can define *map* even more succinctly if we use a point-free style—as usual, the *map*s on sums and products are denoted $(+)$ and $(\times)$.

$$
\begin{array}{lcl}
map\langle\!\langle 1\rangle\!\rangle & = & id \\
map\langle\!\langle Char\rangle\!\rangle & = & id \\
map\langle\!\langle Int\rangle\!\rangle & = & id \\
map\langle\!\langle +\rangle\!\rangle\ mapA\ mapB & = & mapA + mapB \\
map\langle\!\langle \times\rangle\!\rangle\ mapA\ mapB & = & mapA \times mapB
\end{array}
$$

REMARK 3.9 The *copy* function can be extended to functional types:

$$copy\langle A \to B\rangle\ f \quad = \quad copy\langle B\rangle \cdot f \cdot copy\langle A\rangle.$$

However, in this case we can no longer generalize the type of $copy\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle$ to $Map\langle\mathfrak{T}\rangle\ T\ T$ as $copy_{\to}$ does not have the type $Map\langle\star \to \star \to \star\rangle\ (\to)\ (\to)$. □

3.3.1 Defining generic values

The definition of a generic value in MPC-style consists of two parts: a type signature, which typically involves a kind-indexed type, and a set of equations, one for each type constant. Likewise, the definition of a kind-indexed type consists of two parts: a kind signature and one equation for kind $\star$. The equations for product kinds and functional kinds need not be explicitly specified. They are inevitable because of the way type constructors of kind $\mathfrak{T}_1 \times \mathfrak{T}_2$ and $\mathfrak{T}_1 \to \mathfrak{T}_2$ are specialized. In general, a kind-indexed type definition has the following schematic form.

$$
\begin{array}{lcl}
Poly\langle\mathfrak{T} :: \square\rangle & :: & \mathfrak{T} \to \cdots \to \mathfrak{T} \to \star \\
Poly\langle\star\rangle\ T_1\ \ldots\ T_n & = & \\
Poly\langle\mathfrak{A} \times \mathfrak{B}\rangle\ T_1\ \ldots\ T_n & = & Poly\langle\mathfrak{A}\rangle\ (Outl\ T_1)\ \ldots\ (Outl\ T_n) \\
 & & \times\ Poly\langle\mathfrak{B}\rangle\ (Outr\ T_1)\ \ldots\ (Outr\ T_n) \\
Poly\langle\mathfrak{A} \to \mathfrak{B}\rangle\ T_1\ \ldots\ T_n & = & \forall A_1\ \ldots\ A_n\,.\,Poly\langle\mathfrak{A}\rangle\ A_1\ \ldots\ A_n \\
 & & \qquad \to Poly\langle\mathfrak{B}\rangle\ (T_1\ A_1)\ \ldots\ (T_n\ A_n)
\end{array}
$$

The kind signature makes precise that the kind-indexed type $Poly\langle\mathfrak{T} :: \square\rangle$ maps n types of kind $\mathfrak{T}$ to a manifest type (for $Map\langle\mathfrak{T} :: \square\rangle$ we had $n = 2$). The generic programmer merely has to fill out the right-hand side of the first equation.

Given the kind-indexed type a generic value definition takes on the following schematic form.

$$
\begin{array}{lcl}
\mathit{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{Poly}\langle \mathfrak{T} \rangle \; T \; \ldots \; T \\
\mathit{poly}\langle\!\langle 1 \rangle\!\rangle & = & \\
\mathit{poly}\langle\!\langle \mathit{Char} \rangle\!\rangle & = & \\
\mathit{poly}\langle\!\langle \mathit{Int} \rangle\!\rangle & = & \\
\mathit{poly}\langle\!\langle + \rangle\!\rangle & = & \\
\mathit{poly}\langle\!\langle \times \rangle\!\rangle & = & \\
\mathit{poly}\langle\!\langle \rightarrow \rangle\!\rangle & = &
\end{array}
$$

Again, the generic programmer has to fill out the right-hand sides. To be well-typed, the $\mathit{poly}\langle\!\langle C :: \mathfrak{C} \rangle\!\rangle$ instances must have type $\mathit{Poly}\langle \mathfrak{C} \rangle \; C \; \ldots \; C$ as stated in the type signature. As usual, we do not require that an equation is provided for every type constant C in *Const*. In case an equation for C is missing, we tacitly add $\mathit{poly}\langle\!\langle C \rangle\!\rangle = \mathit{undefined}$.

It is worth noting that there are no restrictions on the set *Const* of type constants. In particular, type constants are *not* restricted to types of first-order kind.

3.3.2 Specializing generic values

The type signature of a generic value determines the type for closed type indices. However, since the specialization is defined by induction on the structure of type terms, we must also explicate the type for type indices that contain free type variables. To motivate the necessary amendments let us take a look at an example first. Consider specializing *map* for the type *Matrix* given by $\Lambda A \,.\, \mathit{List}\;(\mathit{List}\;A)$. The definition of $\mathit{map}_{\mathit{Matrix}}$ is

$$
\begin{array}{lcl}
\mathit{map}_{\mathit{Matrix}} & :: & \forall A_1 \; A_2 .\, (A_1 \rightarrow A_2) \rightarrow (\mathit{Matrix}\; A_1 \rightarrow \mathit{Matrix}\; A_2) \\
\mathit{map}_{\mathit{Matrix}} & = & \lambda A_1 \; A_2 .\, \lambda \mathit{map}_A :: (A_1 \rightarrow A_2) \,.\, \mathit{map}_{\mathit{List}}\; (\mathit{List}\; A_1)\; (\mathit{List}\; A_2) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{map}_{\mathit{List}}\; A_1 \; A_2 \; \mathit{map}_A).
\end{array}
$$

First of all, the type of $\mathit{map}_{\mathit{Matrix}}$ determines the type of map_A, which is given by $\mathit{Map}\langle\star\rangle\; A_1 \; A_2 = A_1 \rightarrow A_2$. Now, *Matrix* contains the type term $\mathit{List}\; A$, in which A occurs free. The corresponding mapping function is $\mathit{map}_{\mathit{List}}\; A_1\; A_2\; \mathit{map}_A$, which has type $\mathit{Map}\langle\star\rangle\; (\mathit{List}\; A_1)\; (\mathit{List}\; A_2) = \mathit{List}\; A_1 \rightarrow \mathit{List}\; A_2$. In general, $\mathit{poly}\langle\!\langle T {::} \mathfrak{T} \rangle\!\rangle$ has type $\mathit{Poly}\langle \mathfrak{T} \rangle\; \lfloor T \rfloor_1 \; \ldots \; \lfloor T \rfloor_n$ where $\lfloor T \rfloor_i$ denotes the type term T, in which every free type variable A has been replaced by A_i. To make this work we assume that the individual variable poly_A associated with A has type $\mathit{Poly}\langle \mathfrak{A} \rangle\; A_1 \; \ldots \; A_n$ with $\mathfrak{A} = \mathrm{kind}\; A$ and that the A_i are fresh type variables associated with A. Given these prerequisites the extension of *poly* is defined by

$$
\begin{array}{lcl}
\mathit{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{Poly}\langle \mathfrak{T} \rangle\; \lfloor T \rfloor_1 \; \ldots \; \lfloor T \rfloor_n \\
\mathit{poly}\langle\!\langle A :: \mathfrak{A} \rangle\!\rangle & = & \mathit{poly}_A \\
\mathit{poly}\langle\!\langle (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle & = & (\mathit{poly}\langle\!\langle T_1 :: \mathfrak{T}_1 \rangle\!\rangle, \mathit{poly}\langle\!\langle T_2 :: \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Outl}\; T :: \mathfrak{T}_1 \rangle\!\rangle & = & \mathit{outl}\; (\mathit{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Outr}\; T :: \mathfrak{T}_2 \rangle\!\rangle & = & \mathit{outr}\; (\mathit{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2 \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle (\Lambda A \,.\, T) :: (\mathfrak{S} \rightarrow \mathfrak{T}) \rangle\!\rangle & = & \lambda A_1 \; \ldots \; A_n \,.\, \lambda \mathit{poly}_A \,.\, \mathit{poly}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle \\
\mathit{poly}\langle\!\langle T \; U :: \mathfrak{V} \rangle\!\rangle & = & (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \rightarrow \mathfrak{V} \rangle\!\rangle)\; \lfloor U \rfloor_1 \; \ldots \; \lfloor U \rfloor_n \; (\mathit{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle) \\
\mathit{poly}\langle\!\langle \mathit{Fix}\; T :: \mathfrak{U} \rangle\!\rangle & = & \mathit{fix}\; ((\mathit{poly}\langle\!\langle T :: \mathfrak{U} \rightarrow \mathfrak{U} \rangle\!\rangle)\; \lfloor \mathit{Fix}\; T \rfloor_1 \; \ldots \; \lfloor \mathit{Fix}\; T \rfloor_n)
\end{array}
$$

For $n = 1$ we obtain the definition given in Section 3.1.3. The following theorem states that $\mathit{poly}\langle\!\langle - \rangle\!\rangle$ thus defined is well-typed.

THEOREM 3.10 If $\mathit{poly}\langle\!\langle C :: \mathfrak{C} \rangle\!\rangle :: \mathit{Poly}\langle \mathfrak{C} \rangle\; C \; \ldots \; C$ for all type constants $C \in \mathit{Const}$, then $\mathit{poly}\langle\!\langle X \rangle\!\rangle :: \mathit{Poly}\langle \mathfrak{X} \rangle\; \lfloor X \rfloor_1 \; \ldots \; \lfloor X \rfloor_n$ for all monomorphic type terms $X \in \mathit{MonoType}$ of kind $\mathfrak{X}$.

PROOF. We proceed by induction on the kinding derivation of $X :: \mathfrak{X}$.

- **Case** $X = C :: \mathfrak{C}$: by assumption

$$poly_C :: Poly\langle\mathfrak{C}\rangle\ C\ \dots\ C = Poly\langle\mathfrak{C}\rangle\ \lfloor C\rfloor_1\ \dots\ \lfloor C\rfloor_n.$$

- **Case** $X = A :: \mathfrak{A}$: by assumption

$$poly_A :: Poly\langle\mathfrak{T}\rangle\ A_1\ \dots\ A_n = Poly\langle\mathfrak{T}\rangle\ \lfloor A\rfloor_1\ \dots\ \lfloor A\rfloor_n.$$

- **Case** $X = (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2$: by the induction assumption we have

$$\begin{aligned} poly\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle &:: Poly\langle\mathfrak{T}_1\rangle\ \lfloor T_1\rfloor_1\ \dots\ \lfloor T_1\rfloor_n \text{ and} \\ poly\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle &:: Poly\langle\mathfrak{T}_2\rangle\ \lfloor T_2\rfloor_1\ \dots\ \lfloor T_2\rfloor_n \end{aligned}$$

  and consequently by ($\times$-INTRO)

$$\begin{aligned} &(poly\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle, poly\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle) \\ &\quad :: Poly\langle\mathfrak{T}_1\rangle\ \lfloor T_1\rfloor_1\ \dots\ \lfloor T_1\rfloor_n \times Poly\langle\mathfrak{T}_2\rangle\ \lfloor T_2\rfloor_1\ \dots\ \lfloor T_2\rfloor_n \end{aligned}$$

  Noting that $Outl\ (T_1, T_2) \approx T_1$, $Outr\ (T_1, T_2) \approx T_2$ and $\lfloor(T_1, T_2)\rfloor_i = (\lfloor T_1\rfloor_i, \lfloor T_2\rfloor_i)$ we have

$$\begin{aligned} &Poly\langle\mathfrak{T}_1\rangle\ \lfloor T_1\rfloor_1\ \dots\ \lfloor T_1\rfloor_n \times Poly\langle\mathfrak{T}_2\rangle\ \lfloor T_2\rfloor_1\ \dots\ \lfloor T_2\rfloor_n \\ &\quad \approx Poly\langle\mathfrak{T} \times \mathfrak{U}\rangle\ \lfloor(T_1, T_2)\rfloor_1\ \dots\ \lfloor(T_1, T_2)\rfloor_n. \end{aligned}$$

  Using (CONV) the proposition follows.

- **Case** $X = Outl\ T :: \mathfrak{T}_1$: by the induction assumption we have

$$poly\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle :: Poly\langle\mathfrak{T}_1 \times \mathfrak{T}_2\rangle\ \lfloor T\rfloor_1\ \dots\ \lfloor T\rfloor_n$$

  where

$$\begin{aligned} &Poly\langle\mathfrak{T}_1 \times \mathfrak{T}_2\rangle\ \lfloor T\rfloor_1\ \dots\ \lfloor T\rfloor_n \\ &\quad = Poly\langle\mathfrak{A}\rangle\ (Outl\ \lfloor T\rfloor_1)\ \dots\ (Outl\ \lfloor T\rfloor_n) \\ &\quad\quad \times Poly\langle\mathfrak{B}\rangle\ (Outr\ \lfloor T\rfloor_1)\ \dots\ (Outr\ \lfloor T\rfloor_n) \end{aligned}$$

  Applying ($\times$-ELIM-L) we obtain

$$outl\ (poly\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle) :: Poly\langle\mathfrak{A}\rangle\ (Outl\ \lfloor T\rfloor_1)\ \dots\ (Outl\ \lfloor T\rfloor_n)$$

  Since $\lfloor Outl\ T\rfloor_i = Outl\ \lfloor T\rfloor_i$ the proposition follows.

- **Case** $X = Outr\ T :: \mathfrak{T}_2$: analogous.

- **Case** $X = (\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}$: by the induction assumption we have

$$poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle :: Poly\langle\mathfrak{T}\rangle\ \lfloor T\rfloor_1\ \dots\ \lfloor T\rfloor_n$$

  Using ($\to$-INTRO) and ($\forall$-INTRO) we have

$$\begin{aligned} &\lambda A_1\ \dots\ A_n\,.\,\lambda poly_A\,.\,poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle \\ &\quad :: \forall A_1\ \dots\ A_n\,.\,Poly\langle\mathfrak{S}\rangle\ A_1\ \dots\ A_n \\ &\quad\quad\quad \to Poly\langle\mathfrak{T}\rangle\ \lfloor T\rfloor_1\ \dots\ \lfloor T\rfloor_n. \end{aligned}$$

  Since $\lfloor \Lambda A\,.\,T\rfloor_i\ A_i = \lfloor T\rfloor_i$ the proposition follows.

- **Case** $X = T\ U :: \mathfrak{V}$: by the induction assumption we have

$$\begin{array}{lcl} \mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle & :: & \mathit{Poly}\langle \mathfrak{U} \to \mathfrak{V} \rangle\ \lfloor T \rfloor_1\ \ldots\ \lfloor T \rfloor_n \\ \mathit{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle & :: & \mathit{Poly}\langle \mathfrak{U} \rangle\ \lfloor U \rfloor_1\ \ldots\ \lfloor U \rfloor_n \end{array}$$

where

$$\begin{array}{l} \mathit{Poly}\langle \mathfrak{U} \to \mathfrak{V} \rangle\ \lfloor T \rfloor_1\ \ldots\ \lfloor T \rfloor_n \\ \quad = \quad \forall A_1\ \ldots\ A_n\,.\,\mathit{Poly}\langle \mathfrak{U} \rangle\ A_1\ \ldots\ A_n \\ \qquad\qquad\qquad\qquad \to \mathit{Poly}\langle \mathfrak{V} \rangle\ (\lfloor T \rfloor_1\ A_1)\ \ldots\ (\lfloor T \rfloor_n\ A_n) \end{array}$$

Using ($\forall$-ELIM) we obtain

$$\begin{array}{l} (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle)\ \lfloor U \rfloor_1\ \ldots\ \lfloor U \rfloor_n \\ \quad ::\ \ \mathit{Poly}\langle \mathfrak{U} \rangle\ \lfloor U \rfloor_1\ \ldots\ \lfloor U \rfloor_n \to \mathit{Poly}\langle \mathfrak{V} \rangle\ (\lfloor T \rfloor_1\ \lfloor U \rfloor_1)\ \ldots\ (\lfloor T \rfloor_n\ \lfloor U \rfloor_n) \end{array}$$

and using ($\to$-ELIM)

$$\begin{array}{l} (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V} \rangle\!\rangle)\ \lfloor U \rfloor_1\ \ldots\ \lfloor U \rfloor_n\ (\mathit{poly}\langle\!\langle U :: \mathfrak{U} \rangle\!\rangle) \\ \quad ::\ \ \mathit{Poly}\langle \mathfrak{V} \rangle\ (\lfloor T \rfloor_1\ \lfloor U \rfloor_1)\ \ldots\ (\lfloor T \rfloor_n\ \lfloor U \rfloor_n) \end{array}$$

Since $\lfloor T \rfloor_i\ \lfloor U \rfloor_i = \lfloor T\ U \rfloor_i$ the proposition follows.

- **Case** $X = \mathit{Fix}\ T :: \mathfrak{U}$: by the induction assumption we have

$$\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle \quad :: \quad \mathit{Poly}\langle \mathfrak{U} \to \mathfrak{U} \rangle\ \lfloor T \rfloor_1\ \ldots\ \lfloor T \rfloor_n$$

where

$$\begin{array}{l} \mathit{Poly}\langle \mathfrak{U} \to \mathfrak{U} \rangle\ \lfloor T \rfloor_1\ \ldots\ \lfloor T \rfloor_n \\ \quad = \quad \forall A_1\ \ldots\ A_n\,.\,\mathit{Poly}\langle \mathfrak{U} \rangle\ A_1\ \ldots\ A_n \\ \qquad\qquad\qquad\qquad \to \mathit{Poly}\langle \mathfrak{U} \rangle\ (\lfloor T \rfloor_1\ A_1)\ \ldots\ (\lfloor T \rfloor_n\ A_n) \end{array}$$

Using ($\forall$-ELIM) we obtain

$$\begin{array}{l} (\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle)\ \lfloor \mathit{Fix}\ T \rfloor_1\ \ldots\ \lfloor \mathit{Fix}\ T \rfloor_n \\ \quad ::\ \ \mathit{Poly}\langle \mathfrak{U} \rangle\ \lfloor \mathit{Fix}\ T \rfloor_1\ \ldots\ \lfloor \mathit{Fix}\ T \rfloor_n \\ \qquad \to \mathit{Poly}\langle \mathfrak{U} \rangle\ (\lfloor T \rfloor_1\ \lfloor \mathit{Fix}\ T \rfloor_1)\ \ldots\ (\lfloor T \rfloor_n\ \lfloor \mathit{Fix}\ T \rfloor_n) \end{array}$$

Since $\lfloor \mathit{Fix}\ T \rfloor_i \approx \lfloor T \rfloor_i\ \lfloor \mathit{Fix}\ T \rfloor_i$ we can apply (CONV) and (FIX) to obtain

$$\begin{array}{l} \mathit{fix}\ ((\mathit{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle)\ \lfloor \mathit{Fix}\ T \rfloor_1\ \ldots\ \lfloor \mathit{Fix}\ T \rfloor_n) \\ \quad ::\ \ \mathit{Poly}\langle \mathfrak{U} \rangle\ \lfloor \mathit{Fix}\ T \rfloor_1\ \ldots\ \lfloor \mathit{Fix}\ T \rfloor_n \end{array}$$

as desired. □

Let us conclude the section by noting a trivial consequence of the specialization. Since the structure of types is reflected on the value level, we have $\mathit{poly}\langle\!\langle \Lambda A\,.\,F\ (G\ A) \rangle\!\rangle = \lambda \mathit{poly}_A\,.\,\mathit{poly}\langle\!\langle F \rangle\!\rangle\ (\mathit{poly}\langle\!\langle G \rangle\!\rangle\ \mathit{poly}_A)$. This implies, in particular, that $\mathit{map}\langle\!\langle F \cdot G \rangle\!\rangle = \mathit{map}\langle\!\langle F \rangle\!\rangle \cdot \mathit{map}\langle\!\langle G \rangle\!\rangle$. Perhaps surprisingly, this relationship holds for all generic values, not only for mapping functions. A similar observation is that $\mathit{poly}\langle\!\langle \Lambda A\,.\,A \rangle\!\rangle = \lambda \mathit{poly}_A\,.\,\mathit{poly}_A$ for all generic values. We have, in particular, that $\mathit{map}\langle\!\langle \mathit{Id} \rangle\!\rangle = \mathit{id}$. As an aside, note that these generic identities are not to be confused with the functorial laws $\mathit{map}\langle T \rangle\ \mathit{id} = \mathit{id}$ and $\mathit{map}\langle T \rangle\ (f \cdot g) = \mathit{map}\langle T \rangle\ f \cdot \mathit{map}\langle T \rangle\ g$ (see Section 2.2.2), which are base-level identities.

3.3.3 Examples

Can we turn the generic functions we have encountered so far into MPC-style? The answer is an emphatic "Yes!".

Consider the generic equality function defined in the introduction of Section 3.1. The kind-indexed equality type is

$$
\begin{array}{lcl}
\mathit{Equal}\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \star \\
\mathit{Equal}\langle \star \rangle\ T & = & T \to T \to \mathit{Bool} \\
\mathit{Equal}\langle \mathfrak{A} \times \mathfrak{B} \rangle\ T & = & \mathit{Equal}\langle \mathfrak{A} \rangle\ (\mathit{Outl}\ T) \times \mathit{Equal}\langle \mathfrak{B} \rangle\ (\mathit{Outr}\ T) \\
\mathit{Equal}\langle \mathfrak{A} \to \mathfrak{B} \rangle\ T & = & \forall A\,.\,\mathit{Equal}\langle \mathfrak{A} \rangle\ A \to \mathit{Equal}\langle \mathfrak{B} \rangle\ (T\ A).
\end{array}
$$

Rewriting the POPL-style definition of *equal* into MPC-style is straightforward.

$$
\begin{array}{lcl}
\mathit{equal}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{Equal}\langle \mathfrak{T} \rangle\ T \\
\mathit{equal}\langle\!\langle 1 \rangle\!\rangle\ u_1\ u_2 & = & \mathit{true} \\
\mathit{equal}\langle\!\langle \mathit{Char} \rangle\!\rangle\ c_1\ c_2 & = & \mathit{equalChar}\ c_1\ c_2 \\
\mathit{equal}\langle\!\langle \mathit{Int} \rangle\!\rangle\ i_1\ i_2 & = & \mathit{equalInt}\ i_1\ i_2 \\
\mathit{equal}\langle\!\langle + \rangle\!\rangle\ \mathit{equala}\ \mathit{equalb}\ (\mathit{inl}\ a_1)\ (\mathit{inl}\ a_2) & = & \mathit{equala}\ a_1\ a_2 \\
\mathit{equal}\langle\!\langle + \rangle\!\rangle\ \mathit{equala}\ \mathit{equalb}\ (\mathit{inl}\ a_1)\ (\mathit{inr}\ b_2) & = & \mathit{false} \\
\mathit{equal}\langle\!\langle + \rangle\!\rangle\ \mathit{equala}\ \mathit{equalb}\ (\mathit{inr}\ b_1)\ (\mathit{inl}\ a_2) & = & \mathit{false} \\
\mathit{equal}\langle\!\langle + \rangle\!\rangle\ \mathit{equala}\ \mathit{equalb}\ (\mathit{inr}\ b_2)\ (\mathit{inr}\ b_2) & = & \mathit{equalb}\ b_1\ b_2 \\
\mathit{equal}\langle\!\langle \times \rangle\!\rangle\ \mathit{equala}\ \mathit{equalb}\ (a_1, b_1)\ (a_2, b_2) & = & \mathit{equala}\ a_1\ a_2 \wedge \mathit{equalb}\ b_1\ b_2
\end{array}
$$

Now, since *equal* has a kind-indexed type we can also specialize it for, say, unary type constructors.

$$
\mathit{equal}\langle\!\langle F :: \star \to \star \rangle\!\rangle \quad :: \quad \forall A\,.\,(A \to A \to \mathit{Bool}) \to (F\ A \to F\ A \to \mathit{Bool})
$$

This gives us an extra degree of flexibility: $\mathit{equal}\langle\!\langle F \rangle\!\rangle\ \mathit{op}\ x_1\ x_2$ checks whether corresponding elements in x_1 and x_2 are related by *op*. Of course, *op* need not be an equality operator. PolyLib (Jansson and Jeuring 1998) defines an analogous function but with a more general type:

$$
\mathit{pequal}\langle F :: \star \to \star \rangle \quad :: \quad \forall A_1\ A_2\,.\,(A_1 \to A_2 \to \mathit{Bool}) \to (F\ A_1 \to F\ A_2 \to \mathit{Bool}).
$$

Here, the element types need not be identical. And, in fact, $\mathit{equal}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle$ can be assigned the more general type $\mathit{PEqual}\langle \mathfrak{T} \rangle\ T\ T$ given by

$$
\begin{array}{lcl}
\mathit{PEqual}\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \mathfrak{T} \to \star \\
\mathit{PEqual}\langle \star \rangle\ T_1\ T_2 & = & T_1 \to T_2 \to \mathit{Bool} \\
\mathit{PEqual}\langle \mathfrak{A} \times \mathfrak{B} \rangle\ T_1\ T_2 & = & \mathit{PEqual}\langle \mathfrak{A} \rangle\ (\mathit{Outl}\ T_1)\ (\mathit{Outl}\ T_2) \times \mathit{PEqual}\langle \mathfrak{B} \rangle\ (\mathit{Outr}\ T_1)\ (\mathit{Outr}\ T_2) \\
\mathit{PEqual}\langle \mathfrak{A} \to \mathfrak{B} \rangle\ T_1\ T_2 & = & \forall A_1\ A_2\,.\,\mathit{PEqual}\langle \mathfrak{A} \rangle\ A_1\ A_2 \to \mathit{PEqual}\langle \mathfrak{B} \rangle\ (T_1\ A_1)\ (T_2\ A_2),
\end{array}
$$

which gives us an even greater degree of flexibility.

In general, $\star$-indexed definitions can be easily adopted to MPC-style. Sometimes we can even generalize the types to make the functions more general.

Now, let us turn to a $(\star \to \star)$-indexed value, the *size* function introduced in

Section 1.1.2. Its MPC-style variant, which we call *count*, is given by

$$
\begin{array}{lcl}
\mathit{Count}\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \star \\
\mathit{Count}\langle \star \rangle\ T & = & T \to \mathit{Int} \\
\mathit{Count}\langle \mathfrak{T} \times \mathfrak{U} \rangle\ T & = & \mathit{Count}\langle \mathfrak{T} \rangle\ (\mathit{Outl}\ T) \times \mathit{Count}\langle \mathfrak{U} \rangle\ (\mathit{Outr}\ T) \\
\mathit{Count}\langle \mathfrak{T} \to \mathfrak{U} \rangle\ T & = & \forall A \,.\, \mathit{Count}\langle \mathfrak{T} \rangle\ A \to \mathit{Count}\langle \mathfrak{U} \rangle\ (T\ A) \\
\mathit{count}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{Count}\langle \mathfrak{T} \rangle\ T \\
\mathit{count}\langle\!\langle 1 \rangle\!\rangle\ u & = & 0 \\
\mathit{count}\langle\!\langle \mathit{Char} \rangle\!\rangle\ c & = & 0 \\
\mathit{count}\langle\!\langle \mathit{Int} \rangle\!\rangle\ i & = & 0 \\
\mathit{count}\langle\!\langle + \rangle\!\rangle\ \mathit{countA}\ \mathit{countB}\ (\mathit{inl}\ a) & = & \mathit{countA}\ a \\
\mathit{count}\langle\!\langle + \rangle\!\rangle\ \mathit{countA}\ \mathit{countB}\ (\mathit{inr}\ b) & = & \mathit{countB}\ b \\
\mathit{count}\langle\!\langle \times \rangle\!\rangle\ \mathit{countA}\ \mathit{countB}\ (a, b) & = & \mathit{countA}\ a + \mathit{countB}\ b.
\end{array}
$$

It is not hard to see that $\mathit{count}\langle\!\langle T \rangle\!\rangle\ t$ returns 0 for all types T of kind $\star$ provided t is finite and fully defined (we will prove this in Section 4.3.2). So one might be led to conclude that *count* is not a very useful function. This conclusion is, however, too rash since *count* can also be parameterized by type constructors. For instance, for unary type constructors *count* has type

$$\mathit{count}\langle\!\langle F :: \star \to \star \rangle\!\rangle \quad :: \quad \forall A \,.\, (A \to \mathit{Int}) \to (F\ A \to \mathit{Int}).$$

Now, if we pass the identity function to *count*, we obtain a function that sums up a structure of integers. Another viable choice is $k\ 1$; this yields the *size* function.

$$
\begin{array}{lcl}
\mathit{sum}\langle F :: \star \to \star \rangle & :: & F\ \mathit{Int} \to \mathit{Int} \\
\mathit{sum}\langle F \rangle & = & \mathit{count}\langle\!\langle F \rangle\!\rangle\ \mathit{id} \\
\mathit{size}\langle F :: \star \to \star \rangle & :: & \forall A \,.\, F\ A \to \mathit{Int} \\
\mathit{size}\langle F \rangle & = & \mathit{count}\langle\!\langle F \rangle\!\rangle\ (k\ 1)
\end{array}
$$

In the introduction to Section 3.3 we have discussed how to define mapping functions for types of arbitrary kinds. Interestingly, the MPC-style *map* even subsumes higher-order mapping functions. Recall from Section 3.2.2 that a higher-order mapping function has type $\forall F_1\ F_2 \,.\, (F_1 \dot{\to} F_2) \to (H\ F_1 \dot{\to} H\ F_2)$. Now, the MPC-style map gives us a function of type

$$
\begin{array}{lcl}
\mathit{map}\langle H \rangle & :: & \forall F_1\ F_2 \,.\, (\forall B_1\ B_2 \,.\, (B_1 \to B_2) \to (F_1\ B_1 \to F_2\ B_2)) \\
& & \qquad \to (\forall A_1\ A_2 \,.\, (A_1 \to A_2) \to (H\ F_1\ A_1 \to H\ F_2\ A_2)).
\end{array}
$$

Given a natural transformation m of type $F_1 \dot{\to} F_2$ there are basically two alternatives for constructing a function of type $\forall B_1\ B_2 \,.\, (B_1 \to B_2) \to (F_1\ B_1 \to F_2\ B_2)$: $\lambda h \,.\, m \cdot \mathit{map}_{F_1}\ h$ or $\lambda h \,.\, \mathit{map}_{F_2}\ h \cdot m$. The naturality of m, however, implies that both alternatives are equal. Consequently, the higher-order map is given by

$$
\begin{array}{lcl}
\mathit{hmap}\langle H :: (\star \to \star) \to \star \to \star \rangle & :: & \forall F_1\ F_2 \,.\, (\mathit{Functor}\ F_1, \mathit{Functor}\ F_2) \\
& & \qquad \Rightarrow (F_1 \dot{\to} F_2) \to (H\ F_1 \dot{\to} H\ F_2) \\
\mathit{hmap}\langle H \rangle\ m & = & \mathit{map}\langle\!\langle H \rangle\!\rangle\ (\lambda h \,.\, m \cdot \mathit{fmap}\ h)\ \mathit{id}.
\end{array}
$$

Let us conclude the section with a brief account of the pros and cons of POPL-style and MPC-style definitions. It is undoubtedly easier to write POPL-style definitions (at least if the type index has a first-order kind). MPC-style definitions require more understanding on the user's side but as a compensation they are much more general. If I tackle a generic problem, I usually start writing a POPL-style definition. If it works, I convert it in a second step to MPC-style. We have already seen that $\star$-indexed definitions can be easily adopted to MPC-style. Sometimes one can even generalize the types to make the functions more general. The adaptation of $(\star \to \star)$-indexed functions such as *map* or *size* is not entirely straightforward. It often requires some additional thoughts to be able to formulate a suitable kind-indexed type.

3.4 Related work

Generic programming The concept of generic functional programming appears under a variety of names: Ruehr refers to this concept as *structural polymorphism* (1992, 1998), Sheard calls generic functions *type parametric* (1993), Jay and Cocket use the term *shape polymorphism* (1994), Harper and Morrisett (1995) coined the phrase *intensional polymorphism*, and Jeuring invented the word *polytypism* (1996).

The mainstream of generic programming is based on the initial algebra semantics of data types, see, for instance (Hagino 1987), and puts emphasis on general recursion operators like *map* and catamorphisms (folds). In (Sheard 1991) several variations of these operators are informally defined and algorithms are given that specialize these functions for given data types. The programming language *Charity* (Cockett and Fukushima 1992) automatically provides *map* and catamorphisms for each user-defined data type. Since general recursion is not available, Charity is strongly normalizing. *Functorial ML* (Jay, Bellè, and Moggi 1998) has a similar functionality, but a different background. It is based on the theory of *shape polymorphism*, in which values are separated into shape and contents. The polytypic programming language extension *PolyP* (Jansson and Jeuring 1997) offers a special construct for defining generic functions. The generic definitions are similar to POPL-style definitions (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion, see below for a more detailed comparison.

All the approaches are restricted to first-order kinded, regular data types (or even subsets of this class). One notable exception is the work of Ruehr (1992), who presents a higher-order language based on a type system related to ours (only type recursion is missing). Genericity is achieved through the use of type patterns which are interpreted at run-time. By contrast, the specialization technique presented in Section 3.1.3 does not require the passing of types or representations of types at run-time. This also distinguishes our approach from the work on intensional polymorphism (Harper and Morrisett 1995; Crary, Weirich, and Morrisett 1999) where a **typecase** is used for defining type-dependent operations.

The idea to assign kind-indexed types to type-indexed values is, to the best of the author's knowledge, original. Other approaches to generic programming are restricted in that they only allowed to parameterize values by types of one fixed kind. Three notable exceptions are Functorial ML (Jay, Bellè, and Moggi 1998), the work of Ruehr (1992), and the work of Hoogendijk and Backhouse (1997). Functorial ML allows to quantify over functor arities in type schemes (since Functorial ML only handles regular, first-order functors, kinds can be simplified to arities). However, no formal account of this feature is given and the informal description makes use of an infinitary typing rule. Furthermore, the generic definitions based on this extension are rather unwieldy from a notational point of view. Ruehr also restricts type indices to types of one fixed kind. Additional flexibility is, however, gained through the use of a more expressive kind language, which incorporates kind variables. This extension is used to define a higher-order map indexed by types of kind $(\mathfrak{A} \to \star) \to \star$, where $\mathfrak{A}$ is a kind variable. Clearly, this mapping function is subsumed by the MPC-style *map* given in Section 3.3.3. Whether kind polymorphism has other benefits remains to be seen. Finally, definitions of generic values that are indexed by relators of different arities can be found in the work of Hoogendijk and Backhouse (1997) on commuting data types.

PolyP Currently, PolyP (Jansson and Jeuring 1997) is the only *implemented* generic programming extension for Haskell. It is based on the initial algebra semantics of data types, where recursive data types are modeled by fixed points of associated base functors. Functors and bifunctors are formed according to the following grammar.

$$
\begin{array}{lcl}
F & ::= & \mu B \\
B & ::= & K\ T \mid \mathit{Fst} \mid \mathit{Snd} \mid B + B \mid B \times B \mid F \cdot B
\end{array}
$$

The functor μB, which is known as a *type functor*, denotes the unary functor F given as the least solution of the equation $F\ a = B(a, F\ a)$. Generic functions are defined according to the above structure of functors. For instance, in PolyP the generic function $\mathit{size}\langle F\rangle$ is defined as follows—modulo change of notation.

$$
\begin{array}{lcl}
\mathit{size}\langle F\rangle & :: & \forall A\,.\,F\ A \to \mathit{Int} \\
\mathit{size}\langle \mu B\rangle & = & \mathit{cata}\langle \mu B\rangle\ (\mathit{bsize}\langle B\rangle) \\
\mathit{bsize}\langle B\rangle & :: & \forall A\,.\,B\ A\ \mathit{Int} \to \mathit{Int} \\
\mathit{bsize}\langle K\ T\rangle\ x & = & 0 \\
\mathit{bsize}\langle \mathit{Fst}\rangle\ x & = & 1 \\
\mathit{bsize}\langle \mathit{Snd}\rangle\ n & = & n \\
\mathit{bsize}\langle B_1 + B_2\rangle\ (\mathit{inl}\ x_1) & = & \mathit{bsize}\langle B_1\rangle\ x_1 \\
\mathit{bsize}\langle B_1 + B_2\rangle\ (\mathit{inr}\ x_2) & = & \mathit{bsize}\langle B_2\rangle\ x_2 \\
\mathit{bsize}\langle B_1 \times B_2\rangle\ (x_1, x_2) & = & \mathit{bsize}\langle B_1\rangle\ x_1 + \mathit{bsize}\langle B_2\rangle\ x_2 \\
\mathit{bsize}\langle F \cdot B\rangle\ x & = & \mathit{sum}\langle F\rangle\ (\mathit{map}\langle F\rangle\ (\mathit{bsize}\langle B\rangle)\ x)
\end{array}
$$

The program is quite elaborate as compared to the one given in Section 1.1.2: it involves two general combining forms, the catamorphism *cata* and the mapping function *map*, and an auxiliary generic function, *sum*. The disadvantages of the initial algebra approach are fairly obvious. The above definition is redundant: we know that *size* is uniquely defined by its action on constant functors (that is, $\underline{1}$, $\underline{\mathit{Char}}$, $\underline{\mathit{Int}}$), *Id*, sums, and products. The definition is incomplete: *size* is only applicable to regular functors (recall that, for instance, *Perfect* is not a regular functor). Furthermore, the regular functor may not depend on functors of arity $\geqslant 2$ since functor composition is only defined for unary functors. Finally, the definition exhibits a slight inefficiency: the combing form *map* produces an intermediate data structure, which is immediately consumed by *sum*.

Chapter 4

Generic proofs

If you want to prove a property of a generic value, you have to reason generically. Like the program the proof will be parametric in the underlying data type. This chapter introduces two fundamental generic proof methods. The first method, a variant of fixed point induction, is tailored to POPL-style definitions and proceeds by induction on the structure of types (Section 4.1). Varying the method slightly we can also use it constructively to derive a generic program from its specification (Section 4.2). The second method, which is based on logical relations, generalizes the first method much in the same way as MPC-style definitions generalize POPL-style definitions (Section 4.3). Using a kind-indexed logical relation we prove, for instance, that the generic mapping function satisfies suitable generalizations of the functor laws.

4.1 Fixed point induction

Recall that a generic value such as *encode* or *equal* is defined by induction on the structure of its type argument. In order to deal gracefully with type recursion we do not operate on finite type terms directly but on their potentially infinite Böhm trees. For that reason, the basic proof method associated with POPL-style definitions is fixed point induction. Fixed point induction is like ordinary induction except that the property in question must denote a pointed and chain-complete relation.

Section 4.1.1 introduces fixed point induction for type-indexed values and Section 4.1.2 generalizes the proof method to values indexed by types of first- or second-order kinds.

4.1.1 Type-indexed values

The structure of normal forms, see Section 3.1.1, suggests the following induction principle. Let $\mathcal{P}$ be a type-indexed property that denotes a pointed and chain-complete relation. In order to show that $\mathcal{P}$ holds for all types of kind $\star$, it suffices to show that

$$\begin{array}{l}
\mathcal{P}(1) \\
\mathcal{P}(\mathit{Char}) \\
\mathcal{P}(\mathit{Int}) \\
\forall A \,.\, \mathcal{P}(A) \supset \forall B \,.\, \mathcal{P}(B) \supset \mathcal{P}(A + B) \\
\forall A \,.\, \mathcal{P}(A) \supset \forall B \,.\, \mathcal{P}(B) \supset \mathcal{P}(A \times B) \\
\forall A \,.\, \mathcal{P}(A) \supset \forall B \,.\, \mathcal{P}(B) \supset \mathcal{P}(A \to B).
\end{array}$$

To ensure that $\mathcal{P}$ denotes a pointed relation it suffices to show that $\mathcal{P}(0)$ holds, where '0' is the 'empty' or 'bottom' type with $\mathrm{BT}(0) = \Omega$ and $[\![0]\!] = \bot$. Of course, since we are working in a domain-theoretic setting, '0' is not empty but contains $\bot$ as the single element.

It is useful to know under what conditions $\mathcal{P}$ denotes a chain-complete relation. This is the case, for instance, if $\mathcal{P}$ is built from equalities and inequalities using conjunction, disjunction and universal quantification. All the properties we use are of this restricted form.

We have already encountered a generic proof in Section 1.1.1. The proof established the property $\mathcal{I}nv$ given by

$$\mathcal{I}nv(T) \quad \equiv \quad \forall t :: T \,.\, \forall bin :: Bin \,.\, decodes\langle T\rangle\ (encode\langle T\rangle\ t +\!\!+\ bin) = (t, bin) :: T \otimes Bin.$$

Recall that we assumed that we are working in a strict setting. For that reason, we use smash products, $T \otimes Bin$, rather than products in the equation above. Now, since $\mathcal{I}nv$ takes the form of an equation it is chain-complete. The following calculation shows that it is also pointed.

- **Case** $T = 0$ and $t = \bot$:

$$\begin{aligned} & decodes\langle 0\rangle\ (encode\langle 0\rangle\ \bot +\!\!+\ bin) \\ \equiv\quad & \quad \{\ poly \text{ is strict: } poly\langle 0\rangle = \bot\ \} \\ & \bot \\ \equiv\quad & \quad \{\ \text{pairing is strict for smash products}\ \} \\ & (\bot, bin). \end{aligned}$$

4.1.2 Generalizing to first- and second-order kinds

The extension of the induction scheme to type indices of first- or second-order kinds is straightforward. Recall the normal form of types of kind $\mathfrak{P}$ from Section 3.2.3. Let $\mathcal{P}$ be a type-indexed property, which denotes a pointed and chain-complete relation. In order to show that $\mathcal{P}$ holds for all types of kind $\mathfrak{P}$, it suffices to show that

$$\begin{array}{lll} \forall A_1 \,.\, \mathcal{P}(A_1) \supset \cdots \supset \forall A_{m_1} \,.\, \mathcal{P}(A_{m_1}) & \supset & \mathcal{P}(P_1\ A_1\ \ldots\ A_{m_1}) \\ \ldots & & \\ \forall A_1 \,.\, \mathcal{P}(A_1) \supset \cdots \supset \forall A_{m_n} \,.\, \mathcal{P}(A_{m_n}) & \supset & \mathcal{P}(P_n\ A_1\ \ldots\ A_{m_n}) \\ \forall A_1 \,.\, \mathcal{P}(A_1) \supset \cdots \supset \forall A_{k_1} \,.\, \mathcal{P}(A_{k_1}) & \supset & \mathcal{P}(\underline{C_1}\ A_1\ \ldots\ A_{k_1}) \\ \ldots & & \\ \forall A_1 \,.\, \mathcal{P}(A_1) \supset \cdots \supset \forall A_{k_l} \,.\, \mathcal{P}(A_{k_l}) & \supset & \mathcal{P}(\underline{C_l}\ A_1\ \ldots\ A_{k_l}). \end{array}$$

As before, in order to show that $\mathcal{P}$ denotes a pointed relation, we simply have to establish $\mathcal{P}(\underline{0})$.

To illustrate the induction principle let us prove that the mapping function defined in Section 3.2.1 satisfies the two functor laws, so that a type constructor $T :: \star \to \star$ and its mapping function $map\langle T\rangle$ can, in fact, be seen as the object and morphism part of a functor.

***map* preserves identity** This property can be formalized as follows:

$$\mathcal{I}d(T) \quad \equiv \quad \forall A \,.\, map\langle T\rangle\ id = id :: T\ A \to T\ A.$$

Note that we make the type of the equation explicit. This type information will, in fact, be needed in order to show that $\mathcal{I}d$ is pointed. For the proof we use the point-free definition of map given in Section 3.2.1.

- **Case** $T = \underline{0}$:

$$
\begin{array}{ll}
 & map\langle\underline{0}\rangle\ id \\
\equiv & \quad \{\ poly \text{ is strict: } poly\langle\underline{0}\rangle = \bot\ \} \\
 & \bot \\
\equiv & \quad \{\ \bot = id :: 0 \to 0\ \} \\
 & id.
\end{array}
$$

  Note that the last step is only valid, because the type of the equation is restricted to $0 \to 0$ and there is only one function of type $0 \to 0$ (even in Haskell).

- **Case** $T = Id$:

$$
\begin{array}{ll}
 & map\langle Id\rangle\ id \\
\equiv & \quad \{\ \text{definition of } map\langle Id\rangle\ \} \\
 & id.
\end{array}
$$

- **Case** $T = \underline{1}$:

$$
\begin{array}{ll}
 & map\langle\underline{1}\rangle\ id \\
\equiv & \quad \{\ \text{definition of } map\langle\underline{1}\rangle\ \} \\
 & id.
\end{array}
$$

- **Case** $T = \underline{Char}$: analogous.
- **Case** $T = \underline{Int}$: analogous.
- **Case** $T = F \mathbin{\underline{+}} G$:

$$
\begin{array}{ll}
 & map\langle F \mathbin{\underline{+}} G\rangle\ id \\
\equiv & \quad \{\ \text{definition of } map\langle F \mathbin{\underline{+}} G\rangle\ \} \\
 & map\langle F\rangle\ id + map\langle G\rangle\ id \\
\equiv & \quad \{\ \text{ex hypothesi}\ \} \\
 & id + id \\
\equiv & \quad \{\ (+) \text{ functor}\ \} \\
 & id.
\end{array}
$$

- **Case** $T = F \mathbin{\underline{\times}} G$: analogous.

Unsurprisingly, the proof essentially rests on the fact that Id, $\underline{1}$, $\underline{Char}$, $\underline{Int}$, $(+)$ and $(\times)$ are functors (or bifunctors).

***map* preserves composition** This property is given by

$$
\begin{array}{lll}
\mathcal{C}omp(T) & \equiv & \forall A_1\ A_2\ A_3.\ \forall f :: A_2 \to A_3\ .\ \forall g :: A_1 \to A_2\ . \\
 & & \quad map\langle T\rangle\ (f \cdot g) = map\langle T\rangle\ f \cdot map\langle T\rangle\ g :: T\ A_1 \to T\ A_3.
\end{array}
$$

The straightforward proof is left as an exercise to the reader.

A property of ***size*** Recall the function $size\langle T\rangle :: \forall A\,.\, T\ A \to Int$ introduced in Section 1.1.2, which counts the number of values of type A in a given container of type $T\ A$. The definition presented in Section 1.1.2 uses a pointwise style. For the following calculations a point-free style is preferable:

$$
\begin{array}{lcl}
size\langle T\rangle & :: & \forall A\,.\, T\ A \to Int \\
size\langle Id\rangle & = & k\ 1 \\
size\langle K\ C\rangle & = & k\ 0 \\
size\langle F \mathbin{\underline{+}} G\rangle & = & size\langle F\rangle \triangledown size\langle G\rangle \\
size\langle F \mathbin{\underline{\times}} G\rangle & = & plus \cdot (size\langle F\rangle \times size\langle G\rangle),
\end{array}
$$

where $k\ a\ b = a$ and $plus\ a\ b = a + b$. Note that the definition employs a useful abbreviation: the type pattern $K\ C$ where $K\ A\ B = A$ unites the three cases '$\underline{1}$', '$\underline{Char}$' and '$\underline{Int}$'.

We employ the principle of fixed point induction to establish the following property of $size$: if A is a parameterized type comprising only containers of the same size, that is, $size\langle A\rangle = k\ a$, then

$$
size\langle T \cdot A\rangle \quad = \quad times\ a \cdot size\langle T\rangle, \tag{4.1}
$$

where $times\ a\ b = a \times b$. This law can be used, for instance, to derive a logarithmic implementation of $size\langle Perfect\rangle$—the generic instance has a linear running time. Noting that $Perfect = Id \mathbin{\underline{+}} Perfect \cdot Fork$ we reason:

$$
\begin{array}{cl}
& size\langle Perfect\rangle \\
= & \quad \{\ Perfect = Id \mathbin{\underline{+}} Perfect \cdot Fork\ \} \\
& size\langle Id \mathbin{\underline{+}} Perfect \cdot Fork\rangle \\
= & \quad \{\ \text{definition of } size\ \} \\
& k\ 1 \triangledown size\langle Perfect \cdot Fork\rangle \\
= & \quad \{\ \text{property (4.1) and } size\langle Fork\rangle = k\ 2\ \} \\
& k\ 1 \triangledown times\ 2 \cdot size\langle Perfect\rangle.
\end{array}
$$

If we remove the abstract clothing, we obtain the following Haskell program:

$$
\begin{array}{lcl}
sizePerfect & :: & \forall A\,.\, Perfect\ A \to Int \\
sizePerfect\ (zeroP\ a) & = & 1 \\
sizePerfect\ (succP\ p) & = & 2 \times sizePerfect\ p.
\end{array}
$$

Now for the proof of the property:

- **Case** $T = \underline{0}$:

$$
\begin{array}{cl}
& size\langle \underline{0} \cdot A\rangle \\
= & \quad \{\ \underline{0} \cdot A = \underline{0}\ \} \\
& size\langle \underline{0}\rangle \\
= & \quad \{\ poly \text{ is strict: } poly\langle \underline{0}\rangle = \bot\ \} \\
& \bot \\
= & \quad \{\ times\ a \text{ is strict}\ \} \\
& times\ a \cdot \bot \\
= & \quad \{\ poly \text{ is strict: } poly\langle \underline{0}\rangle = \bot\ \} \\
& times\ a \cdot size\langle \underline{0}\rangle.
\end{array}
$$

- **Case** $T = Id$:

$$
\begin{array}{ll}
 & size\langle Id \cdot A\rangle \\
= & \quad \{\ Id \cdot A = A\ \} \\
 & size\langle A\rangle \\
= & \quad \{\ \text{assumption: } size\langle A\rangle = k\ a\ \} \\
 & k\ a \\
= & \quad \{\ \text{arithmetic: } a = a \times 1\ \} \\
 & times\ a \cdot k\ 1 \\
= & \quad \{\ \text{definition of } size\ \} \\
 & times\ a \cdot size\langle Id\rangle.
\end{array}
$$

- **Case** $T = K\ C$:

$$
\begin{array}{ll}
 & size\langle K\ C \cdot A\rangle \\
= & \quad \{\ K\ C \cdot A = K\ C\ \} \\
 & size\langle K\ C\rangle \\
= & \quad \{\ \text{definition of } size\ \} \\
 & k\ 0 \\
= & \quad \{\ \text{arithmetic: } a \times 0 = 0\ \} \\
 & times\ a \cdot k\ 0 \\
= & \quad \{\ \text{definition of } size\ \} \\
 & times\ a \cdot size\langle K\ C\rangle.
\end{array}
$$

- **Case** $T = F \mathbin{\underline{+}} G$:

$$
\begin{array}{ll}
 & size\langle (F \mathbin{\underline{+}} G) \cdot A\rangle \\
= & \quad \{\ (F \mathbin{\underline{+}} G) \cdot A = F \cdot A \mathbin{\underline{+}} G \cdot A\ \} \\
 & size\langle F \cdot A \mathbin{\underline{+}} G \cdot A\rangle \\
= & \quad \{\ \text{definition of } size\ \} \\
 & size\langle F \cdot A\rangle \triangledown size\langle G \cdot A\rangle \\
= & \quad \{\ \text{ex hypothesi}\ \} \\
 & (times\ a \cdot size\langle F\rangle) \triangledown (times\ a \cdot size\langle G\rangle) \\
= & \quad \{\ \triangledown\text{-fusion law: } h \cdot (f \triangledown g) = (h \cdot f) \triangledown (h \cdot g)\ \} \\
 & times\ a \cdot (size\langle F\rangle \triangledown size\langle G\rangle) \\
= & \quad \{\ \text{definition of } size\ \} \\
 & times\ a \cdot size\langle F \mathbin{\underline{+}} G\rangle.
\end{array}
$$

- **Case** $T = F \mathbin{\underline{\times}} G$:

$$
\begin{array}{ll}
 & size\langle (F \mathbin{\underline{\times}} G) \cdot A\rangle \\
= & \quad \{\ (F \mathbin{\underline{\times}} G) \cdot A = F \cdot A \mathbin{\underline{\times}} G \cdot A\ \} \\
 & size\langle F \cdot A \mathbin{\underline{\times}} G \cdot A\rangle \\
= & \quad \{\ \text{definition of } size\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & plus \cdot (size\langle F \cdot A\rangle \times size\langle G \cdot A\rangle) \\
= & \quad \{ \text{ex hypothesi} \} \\
 & plus \cdot ((times\ a \cdot size\langle F\rangle) \times (times\ a \cdot size\langle G\rangle)) \\
= & \quad \{ (\times) \text{ bifunctor} \} \\
 & plus \cdot (times\ a \times times\ a) \cdot (size\langle F\rangle \times size\langle G\rangle) \\
= & \quad \{ \text{arithmetic: } a \times (b + c) = a \times b + a \times c \} \\
 & times\ a \cdot plus \cdot (size\langle F\rangle \times size\langle G\rangle) \\
= & \quad \{ \text{definition of } size \} \\
 & times\ a \cdot size\langle F \underline{\times} G\rangle.
\end{array}
$$

Perhaps unusual, the proof involves both calculations on the value and on the type level. We will encounter more examples of this type in due course.

4.2 Deriving generic programs

In the preceding section we have employed fixed point induction to prove a generic property of a given generic program. Perhaps surprisingly, we can also use the method constructively to *derive* a generic program from a generic specification. Rather than formalizing the technique we introduce it by means of an example: we show how to derive an already known function, namely *decodes*, by inverse function construction. We proceed in two steps.

Deriving *encodes* Reconsider the definition of *encode* given in Section 1.1.1 (on page 8). Since *encode* uses list concatenation, $(+\!\!+)$, to encode a pair of values, it exhibits $\Theta(n^2)$ worst-case behaviour. In a first step we remedy this defect using the well-known technique of *accumulation* (Bird 1984). The basic idea is to define a function that encodes a value and additionally appends a given bit stream to the result:

$$encodes\langle T\rangle\ (t, bin) \quad = \quad encode\langle T\rangle\ t +\!\!+ bin. \tag{4.2}$$

Since $x +\!\!+ [\,] = x$, we can easily define *encode* in terms of the more efficient *encodes*: we have $encode\langle T\rangle\ t = encodes\langle T\rangle\ (t, [\,])$.

Now, since $(+\!\!+)$ is strict in its first argument, the specification holds trivially for $T = 0$. To derive a definition for *encodes* we reason as follows.

- **Case** $T = 1$ and $t = ()$:

$$
\begin{array}{ll}
 & encodes\langle 1\rangle\ ((), bin) \\
= & \quad \{ \text{specification (4.2)} \} \\
 & encode\langle 1\rangle\ () +\!\!+ bin \\
= & \quad \{ \text{definition of } encode \} \\
 & [\,] +\!\!+ bin \\
= & \quad \{ \text{`}[\,]\text{' is the unit of } (+\!\!+)\text{: } [\,] +\!\!+ x = x \} \\
 & bin.
\end{array}
$$

- **Case** $T = A + B$ and $t = inl\ a$:

$$encodes\langle A + B\rangle\ (inl\ a, bin)$$

$$
\begin{aligned}
&= \quad \{\ \text{specification (4.2)}\ \} \\
&\quad \mathit{encode}\langle A + B\rangle\ (\mathit{inl}\ a) \mathbin{+\!\!+} \mathit{bin} \\
&= \quad \{\ \text{definition of } \mathit{encode}\ \} \\
&\quad (\mathtt{0} : \mathit{encode}\langle A\rangle\ a) \mathbin{+\!\!+} \mathit{bin} \\
&= \quad \{\ \text{definition of } (\mathbin{+\!\!+})\text{: } (a : x) \mathbin{+\!\!+} y = a : (x \mathbin{+\!\!+} y)\ \} \\
&\quad \mathtt{0} : (\mathit{encode}\langle A\rangle\ a \mathbin{+\!\!+} \mathit{bin}) \\
&= \quad \{\ \text{specification (4.2)}\ \} \\
&\quad \mathtt{0} : \mathit{encodes}\langle A\rangle\ (a, \mathit{bin}).
\end{aligned}
$$

- **Case** $T = A + B$ and $t = \mathit{inr}\ b$: analogous.
- **Case** $T = A \times B$ and $t = (a, b)$:

$$
\begin{aligned}
&\quad \mathit{encodes}\langle A \times B\rangle\ ((a, b), \mathit{bin}) \\
&= \quad \{\ \text{specification (4.2)}\ \} \\
&\quad \mathit{encode}\langle A \times B\rangle\ (a, b) \mathbin{+\!\!+} \mathit{bin} \\
&= \quad \{\ \text{definition of } \mathit{encode}\ \} \\
&\quad (\mathit{encode}\langle A\rangle\ a \mathbin{+\!\!+} \mathit{encode}\langle B\rangle\ b) \mathbin{+\!\!+} \mathit{bin} \\
&= \quad \{\ (\mathbin{+\!\!+}) \text{ is associative: } (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z = x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z)\ \} \\
&\quad \mathit{encode}\langle A\rangle\ a \mathbin{+\!\!+} (\mathit{encode}\langle B\rangle\ b \mathbin{+\!\!+} \mathit{bin}) \\
&= \quad \{\ \text{specification (4.2)}\ \} \\
&\quad \mathit{encodes}\langle A\rangle\ (a, \mathit{encode}\langle B\rangle\ b \mathbin{+\!\!+} \mathit{bin}) \\
&= \quad \{\ \text{specification (4.2)}\ \} \\
&\quad \mathit{encodes}\langle A\rangle\ (a, \mathit{encodes}\langle B\rangle\ (b, \mathit{bin})).
\end{aligned}
$$

Thus, we have derived the following definition of *encodes*:

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Encodes}\ A & = & A \times \mathit{Bin} \to \mathit{Bin} \\
\mathit{encodes}\langle T :: \star\rangle & :: & \mathit{Encodes}\ T \\
\mathit{encodes}\langle 1\rangle\ ((), \mathit{bin}) & = & \mathit{bin} \\
\mathit{encodes}\langle A + B\rangle\ (\mathit{inl}\ a, \mathit{bin}) & = & \mathtt{0} : \mathit{encodes}\langle A\rangle\ (a, \mathit{bin}) \\
\mathit{encodes}\langle A + B\rangle\ (\mathit{inr}\ b, \mathit{bin}) & = & \mathtt{1} : \mathit{encodes}\langle B\rangle\ (b, \mathit{bin}) \\
\mathit{encodes}\langle A \times B\rangle\ ((a, b), \mathit{bin}) & = & \mathit{encodes}\langle A\rangle\ (a, \mathit{encodes}\langle B\rangle\ (b, \mathit{bin})).
\end{array}
$$

Is the definition correct? Yes, we can easily reorder the derivation to obtain an inductive proof. Note that the derivation has a particular structure: in the first step we apply the specification from left to right; then in later steps we (possibly) apply the specification from right to left, which corresponds to using the induction hypothesis. If a derivation has this characteristic structure, we can always rewrite it into an inductive proof.

Deriving *decodes* Given the definition of *encodes* we can derive *decodes* by inverse function construction:

$$
\mathit{decodes}\langle T\rangle \cdot \mathit{encodes}\langle T\rangle \quad = \quad \mathit{id}. \tag{4.3}
$$

Before we proceed let us first rewrite *encodes* into a point-free style since this allows for more structured calculations. To this end it is useful to define some combinators that operate on bit streams:

$$
\begin{array}{lcl}
\mathit{emit} & :: & \mathit{Bit} \to (\mathit{Bin} \to \mathit{Bin}) \\
\mathit{emit}\ b\ \mathit{bin} & = & b : \mathit{bin} \\
\mathit{switch} & :: & \forall A\,.\,(\mathit{Bin} \to A) \to (\mathit{Bin} \to A) \to (\mathit{Bin} \to A) \\
\mathit{switch}\ f\ g\ (\mathtt{0} : \mathit{bin}) & = & f\ \mathit{bin} \\
\mathit{switch}\ f\ g\ (\mathtt{1} : \mathit{bin}) & = & g\ \mathit{bin}.
\end{array}
$$

Roughly speaking, *switch* is for bit streams what $(\triangledown)$ is for sums and $\mathit{emit}\ \mathtt{0}$ and $\mathit{emit}\ \mathtt{1}$ are the analogues of *inl* and *inr*. The following laws lay down the interaction between sums and bit streams.

$$
\mathit{switch}\ f\ g \cdot (\mathit{emit}\ \mathtt{0} \triangledown \mathit{emit}\ \mathtt{1}) = f \triangledown g \tag{4.4}
$$

$$
(f \triangledown g) \cdot \mathit{switch}\ \mathit{inl}\ \mathit{inr} = \mathit{switch}\ f\ g \tag{4.5}
$$

Actually, it suffices to remember one law. The second is then obtained by systematically exchanging $(\triangledown)$ with *switch*, *inl* with $\mathit{emit}\ \mathtt{0}$, and *inr* with $\mathit{emit}\ \mathtt{1}$.

Now, the point-free definition of *encodes* is given by

$$
\begin{array}{lcl}
\mathit{encodes}\langle T :: \star\rangle & :: & \mathit{Encodes}\ T \\
\mathit{encodes}\langle 1\rangle & = & \mathit{unit} \\
\mathit{encodes}\langle A + B\rangle & = & \mathit{encodes}\langle A\rangle \rightarrowtail\!\!+\!\!\!> \mathit{encodes}\langle B\rangle \\
\mathit{encodes}\langle A \times B\rangle & = & \mathit{encodes}\langle A\rangle \gg\!\!> \mathit{encodes}\langle B\rangle
\end{array}
$$

where

$$
\begin{array}{lcl}
(\rightarrowtail\!\!+\!\!\!>) & :: & \forall A\ B\,.\,\mathit{Encodes}\ A \to \mathit{Encodes}\ B \to \mathit{Encodes}\ (A + B) \\
f \rightarrowtail\!\!+\!\!\!> g & = & ((\mathit{emit}\ \mathtt{0} \cdot f) \triangledown (\mathit{emit}\ \mathtt{1} \cdot g)) \cdot \mathit{distl} \\
(\gg\!\!>) & :: & \forall A\ B\,.\,\mathit{Encodes}\ A \to \mathit{Encodes}\ B \to \mathit{Encodes}\ (A \times B) \\
f \gg\!\!> g & = & f \cdot (\mathit{id} \times g) \cdot \mathit{assocr}.
\end{array}
$$

To reassure you that the two definitions of *encodes* are identical we quickly calculate that

$$
\begin{array}{lcl}
(f \rightarrowtail\!\!+\!\!\!> g)\ (\mathit{inl}\ a, \mathit{bin}) & = & \mathtt{0} : f\ (a, \mathit{bin}) \\
(f \rightarrowtail\!\!+\!\!\!> g)\ (\mathit{inr}\ b, \mathit{bin}) & = & \mathtt{1} : g\ (b, \mathit{bin}) \\
(f \gg\!\!> g)\ ((a, b), \mathit{bin}) & = & f\ (a, g\ (b, \mathit{bin})).
\end{array}
$$

You can think of $(\rightarrowtail\!\!+\!\!\!>)$ and $(\gg\!\!>)$ as combinators for encoding sums and products. As an aside, note that using these combinators we can easily specialize *encodes* to given instances of data types. Take, for example, the *List* instance:

$$
\begin{array}{lcl}
\mathit{encodesList} & :: & \forall A\,.\,\mathit{Encodes}\ A \to \mathit{Encodes}\ (\mathit{List}\ A) \\
\mathit{encodesList}\ \mathit{encodesA} & = & \mathit{unit} \rightarrowtail\!\!+\!\!\!> \mathit{encodesA} \gg\!\!> \mathit{encodesList}\ \mathit{encodesA}.
\end{array}
$$

Now, let us derive *decodes*. Note that the specification (4.3) holds for $T = 0$ since $\perp = \mathit{id} :: 0 \times \mathit{Bin} \to 0 \times \mathit{Bin}$.

- **Case** $T = 1$:

$$
\begin{array}{cl}
 & \mathit{decodes}\langle 1\rangle \cdot \mathit{encodes}\langle 1\rangle = \mathit{id} \\
\equiv & \quad \{\ \text{definition of } \mathit{encodes}\ \} \\
 & \mathit{decodes}\langle 1\rangle \cdot \mathit{unit} = \mathit{id} \\
\equiv & \quad \{\ \mathit{unit} : 1 \times A \cong A : \mathit{ununit}\ \} \\
 & \mathit{decodes}\langle 1\rangle = \mathit{ununit}.
\end{array}
$$

- **Case** $T = A + B$:

$$
\begin{array}{ll}
& \mathit{decodes}\langle A+B\rangle \cdot \mathit{encodes}\langle A+B\rangle = \mathit{id} \\
\equiv & \quad \{ \text{ definition of } \mathit{encodes} \text{ and } (\gg\!+\!\!>) \ \} \\
& \mathit{decodes}\langle A+B\rangle \cdot ((\mathit{emit}\ \mathtt{0} \cdot \mathit{encodes}\langle A\rangle) \triangledown (\mathit{emit}\ \mathtt{1} \cdot \mathit{encodes}\langle B\rangle)) \cdot \mathit{distl} = \mathit{id} \\
\equiv & \quad \{ \ \mathit{distl} : (A+B) \times C \cong (A \times C) + (B \times C) : \mathit{undistl} \ \} \\
& \mathit{decodes}\langle A+B\rangle \cdot ((\mathit{emit}\ \mathtt{0} \cdot \mathit{encodes}\langle A\rangle) \triangledown (\mathit{emit}\ \mathtt{1} \cdot \mathit{encodes}\langle B\rangle)) = \mathit{undistl} \\
\equiv & \quad \{ \ \triangledown\text{-+-fusion law: } (f \triangledown g) \cdot (h+k) = (f \cdot h) \triangledown (g \cdot k) \ \} \\
& \mathit{decodes}\langle A+B\rangle \cdot (\mathit{emit}\ \mathtt{0} \triangledown \mathit{emit}\ \mathtt{1}) \cdot (\mathit{encodes}\langle A\rangle + \mathit{encodes}\langle B\rangle) = \mathit{undistl} \\
\subset & \quad \{ \text{ specification (4.3) and } (+) \text{ bifunctor } \} \\
& \mathit{decodes}\langle A+B\rangle \cdot (\mathit{emit}\ \mathtt{0} \triangledown \mathit{emit}\ \mathtt{1}) = \mathit{undistl} \cdot (\mathit{decodes}\langle A\rangle + \mathit{decodes}\langle B\rangle) \\
\equiv & \quad \{ \text{ reflection law: } \mathit{inl} \triangledown \mathit{inr} = \mathit{id} \ \} \\
& \mathit{decodes}\langle A+B\rangle \cdot (\mathit{emit}\ \mathtt{0} \triangledown \mathit{emit}\ \mathtt{1}) = \mathit{undistl} \cdot (\mathit{decodes}\langle A\rangle + \mathit{decodes}\langle B\rangle) \cdot (\mathit{inl} \triangledown \mathit{inr}) \\
\subset & \quad \{ \text{ property (4.4) } \} \\
& \mathit{decodes}\langle A+B\rangle = \mathit{undistl} \cdot (\mathit{decodes}\langle A\rangle + \mathit{decodes}\langle B\rangle) \cdot \mathit{switch}\ \mathit{inl}\ \mathit{inr} \\
\equiv & \quad \{ \text{ definition of } (+) \ \} \\
& \mathit{decodes}\langle A+B\rangle = \mathit{undistl} \cdot (\mathit{inl} \cdot \mathit{decodes}\langle A\rangle \triangledown \mathit{inr} \cdot \mathit{decodes}\langle B\rangle) \cdot \mathit{switch}\ \mathit{inl}\ \mathit{inr} \\
\equiv & \quad \{ \text{ property (4.5) } \} \\
& \mathit{decodes}\langle A+B\rangle = \mathit{undistl} \cdot \mathit{switch}\ (\mathit{inl} \cdot \mathit{decodes}\langle A\rangle)\ (\mathit{inr} \cdot \mathit{decodes}\langle B\rangle).
\end{array}
$$

- **Case** $T = A \times B$:

$$
\begin{array}{ll}
& \mathit{decodes}\langle A \times B\rangle \cdot \mathit{encodes}\langle A \times B\rangle = \mathit{id} \\
\equiv & \quad \{ \text{ definition of } \mathit{encodes} \text{ and } (\gg\!\!\gg\!\!>) \ \} \\
& \mathit{decodes}\langle A \times B\rangle \cdot \mathit{encodes}\langle A\rangle \cdot (\mathit{id} \times \mathit{encodes}\langle B\rangle) \cdot \mathit{assocr} = \mathit{id} \\
\equiv & \quad \{ \ \mathit{assocl} : A \times (B \times C) \cong (A \times B) \times C : \mathit{assocr} \ \} \\
& \mathit{decodes}\langle A \times B\rangle \cdot \mathit{encodes}\langle A\rangle \cdot (\mathit{id} \times \mathit{encodes}\langle B\rangle) = \mathit{assocl} \\
\subset & \quad \{ \text{ specification (4.3) and } (\times) \text{ bifunctor } \} \\
& \mathit{decodes}\langle A \times B\rangle \cdot \mathit{encodes}\langle A\rangle = \mathit{assocl} \cdot (\mathit{id} \times \mathit{decodes}\langle B\rangle) \\
\subset & \quad \{ \text{ specification (4.3) } \} \\
& \mathit{decodes}\langle A \times B\rangle = \mathit{assocl} \cdot (\mathit{id} \times \mathit{decodes}\langle B\rangle) \cdot \mathit{decodes}\langle A\rangle.
\end{array}
$$

Thus, we obtain the following definition of *decodes*:

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Decodes}\ A & = & \mathit{Bin} \to A \times \mathit{Bin} \\
\mathit{decodes}\langle T :: \star\rangle & :: & \mathit{Decodes}\ T \\
\mathit{decodes}\langle 1\rangle & = & \mathit{ununit} \\
\mathit{decodes}\langle A+B\rangle & = & \mathit{decodes}\langle A\rangle \lessdot\!\!+\!\!< \mathit{decodes}\langle B\rangle \\
\mathit{decodes}\langle A \times B\rangle & = & \mathit{decodes}\langle A\rangle \lessdot\!\!\ll \mathit{decodes}\langle B\rangle,
\end{array}
$$

where

$$
\begin{array}{lcl}
(\lessdot\!\!+\!\!<) & :: & \forall A\ B\,.\ \mathit{Decodes}\ A \to \mathit{Decodes}\ B \to \mathit{Decodes}\ (A+B) \\
f \lessdot\!\!+\!\!< g & = & \mathit{undistl} \cdot \mathit{switch}\ (\mathit{inl} \cdot f)\ (\mathit{inr} \cdot g) \\
(\lessdot\!\!\ll) & :: & \forall A\ B\,.\ \mathit{Decodes}\ A \to \mathit{Decodes}\ B \to \mathit{Decodes}\ (A \times B) \\
f \lessdot\!\!\ll g & = & \mathit{assocl} \cdot (\mathit{id} \times g) \cdot f.
\end{array}
$$

Is this definition of *decodes* equivalent to the one given in Section 1.1.1 (on page 8)? The answer is in the affirmative. The equivalence is easy to see if we rewrite ($\lessdot\!\!+\!\!<$) and ($\lessdot\!\!\times\!\!<$) into a pointwise form:

$$
\begin{array}{lll}
(f \lessdot\!\!+\!\!< g)\ (0 : bin) & = & \mathbf{let}\ (a, bin') = f\ bin\ \mathbf{in}\ (inl\ a, bin') \\
(f \lessdot\!\!+\!\!< g)\ (1 : bin) & = & \mathbf{let}\ (b, bin') = g\ bin\ \mathbf{in}\ (inr\ b, bin') \\
(f \lessdot\!\!\times\!\!< g)\ bin & = & \mathbf{let}\ (a, bin_1) = f\ bin \\
& & \phantom{\mathbf{let}}\ (b, bin_2) = g\ bin_1 \\
& & \mathbf{in}\ ((a, b), bin_2).
\end{array}
$$

Remark 4.1 We have used the pointwise style for the first but the point-free style for the second derivation. Why this change of style? Now, the point-free style is usually preferable for calculations (if you are not convinced, redo the second derivation in a pointwise style). The first derivation is, however, a notable exception to this rule. Note that central use is made of the fact that '[ ]' is the unit of ($+\!\!+$) and that ($+\!\!+$) is associative. These properties are simple to state in a pointwise style

$$
\begin{array}{lll}
[\,] +\!\!+ x & = & x \\
x +\!\!+ (y +\!\!+ z) & = & (x +\!\!+ y) +\!\!+ z
\end{array}
$$

but they are barely recognizable when expressed in a point-free style:

$$
\begin{array}{lll}
cat \cdot (nil \times id) & = & unit \\
cat \cdot (cat \times id) & = & cat \cdot (id \times cat) \cdot assocr,
\end{array}
$$

where $nil :: \forall A\,.\,1 \to [A]$ and $cat :: \forall A\,.\,[A] \to [A] \to [A]$. For a more thorough discussion of pointwise versus point-free reasoning we refer the interested reader to de Moor and Gibbons (2000). □

4.3 Generic logical relations

MPC-style definitions generalize POPL-style definitions in that they allow to parameterize a generic value by types of arbitrary kinds. In much the same way proofs based on generic logical relations generalize inductive proofs. An inductive proof establishes a property that is parameterized by types of one fixed kind. By contrast, a generic logical relation is a kind-indexed family of such properties. Let us introduce the proof technique by means of our running example: mapping functions.

Recall the MPC-style definition of *map* given in Section 3.3 (on page 76). To classify as a functor the mapping function of a unary type constructor must satisfy the functor laws:

$$
\begin{array}{lll}
map\langle\!\langle T\rangle\!\rangle\ id & = & id \\
map\langle\!\langle T\rangle\!\rangle\ (f \cdot g) & = & map\langle\!\langle T\rangle\!\rangle\ f \cdot map\langle\!\langle T\rangle\!\rangle\ g,
\end{array}
$$

that is, $map\langle\!\langle T\rangle\!\rangle$ preserves identity and composition. If the type constructor is binary, the functor laws take the form

$$
\begin{array}{lll}
map\langle\!\langle T\rangle\!\rangle\ id\ id & = & id \\
map\langle\!\langle T\rangle\!\rangle\ (f_1 \cdot f_2)\ (g_1 \cdot g_2) & = & map\langle\!\langle T\rangle\!\rangle\ f_1\ g_1 \cdot map\langle\!\langle T\rangle\!\rangle\ f_2\ g_2.
\end{array}
$$

How can we generalize these laws to data types of arbitrary kinds? Since $map\langle\!\langle T\rangle\!\rangle$ has a kind-indexed type, it is reasonable to expect that the functorial properties are indexed by kinds, as well. So, what form do the laws take if the type index is a manifest type of kind $\star$? In this case $map\langle\!\langle T\rangle\!\rangle$ does not preserve identity; it *is* the identity:

$$\begin{array}{lcl} map\langle\!\langle T\rangle\!\rangle & = & id \\ map\langle\!\langle T\rangle\!\rangle & = & map\langle\!\langle T\rangle\!\rangle \cdot map\langle\!\langle T\rangle\!\rangle. \end{array}$$

The pendant of the second law states that $map\langle\!\langle T\rangle\!\rangle$ is idempotent (which is a simple consequence of the first law). Given this base case the generalization to arbitrary kinds is within reach. The generic version of the first functor law states that $map\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle \in \mathcal{I}d\langle\mathfrak{T}\rangle\ T$ for all closed monomorphic types $T \in MonoType$, where $\mathcal{I}d$ is given by

$$\begin{array}{lcl} \mathcal{I}d\langle\mathfrak{T}\rangle\ T & \subseteq & Map\langle\mathfrak{T}\rangle\ T\ T \\ m \in \mathcal{I}d\langle\star\rangle\ T & \equiv & m = id :: T \to T \\ m \in \mathcal{I}d\langle\mathfrak{A}\times\mathfrak{B}\rangle\ T & \equiv & outl\ m \in \mathcal{I}d\langle\mathfrak{A}\rangle\ (Outl\ T) \cap outr\ m \in \mathcal{I}d\langle\mathfrak{B}\rangle\ (Outr\ T) \\ m \in \mathcal{I}d\langle\mathfrak{A}\to\mathfrak{B}\rangle\ T & \equiv & \forall A :: \mathfrak{A}\,.\,\forall a :: Map\langle\mathfrak{A}\rangle\ A\ A\,.\,a \in \mathcal{I}d\langle\mathfrak{A}\rangle\ A \supset m\ A\ a \in \mathcal{I}d\langle\mathfrak{B}\rangle\ (T\ A). \end{array}$$

The relation $\mathcal{I}d$ strongly resembles a unary logical relation, see Section 2.4.4. The second and the third clause of the definition are characteristic for logical relations; they guarantee that the relation is closed under projection and pairing, and application and abstraction. We will call $\mathcal{I}d$ and its colleagues generic logical relations (or simply logical relations) for want of a better name. Section 4.3.1 details the differences between generic and 'classical' logical relations.

In a similar vein, the generic version of the second functor law expresses that $(map\langle\!\langle T{::}\mathfrak{T}\rangle\!\rangle, map\langle\!\langle T{::}\mathfrak{T}\rangle\!\rangle, map\langle\!\langle T{::}\mathfrak{T}\rangle\!\rangle) \in \mathcal{C}omp\langle\mathfrak{T}\rangle\ T\ T\ T$ for all closed monomorphic types $T \in MonoType$, where $\mathcal{C}omp$ is given by

$$\begin{array}{lcl} \mathcal{C}omp\langle\mathfrak{T}\rangle\ T_1\ T_2\ T_3 & \subseteq & Map\langle\mathfrak{T}\rangle\ T_2\ T_3 \times Map\langle\mathfrak{T}\rangle\ T_1\ T_2 \times Map\langle\mathfrak{T}\rangle\ T_1\ T_3 \\ (m_1, m_2, m_3) \in \mathcal{C}omp\langle\star\rangle\ T_1\ T_2\ T_3 & \equiv & m_1 \cdot m_2 = m_3 :: T_1 \to T_3. \end{array}$$

It is not hard to see that the 'ordinary' functor laws are instances of these generic laws. We have, for instance,

$$\begin{array}{l} mapT \in \mathcal{I}d\langle\star\to\star\to\star\rangle\ T \\ \quad\equiv \forall A :: \star\,.\,\forall mA :: A \to A\,.\,mA = id :: A \to A \\ \qquad\qquad \supset \forall B :: \star\,.\,\forall mB :: B \to B\,.\,mB = id :: B \to B \\ \qquad\qquad\qquad \supset mapT\ A\ mA\ B\ mB = id :: T\ A\ B \to T\ A\ B \\ \quad\equiv \forall A :: \star\,.\,\forall B :: \star\,.\,mapT\ A\ id\ B\ id = id :: T\ A\ B \to T\ A\ B. \end{array}$$

Turning to the proof of the first generic law we must show (i) that $\mathcal{I}d$ is pointed and chain-complete and (ii) $map\langle\!\langle C :: \mathfrak{C}\rangle\!\rangle \in \mathcal{I}d\langle\mathfrak{C}\rangle\ C$ for all type constants $C \in Const$. Now, $\mathcal{I}d$ is chain-complete since the property takes the form of an equation. Pointedness means that $\bot \in \mathcal{I}d\langle\star\rangle\ 0 \equiv \bot = id :: 0 \to 0$. This holds since there is only one function of type $0 \to 0$. The proof of condition (ii) is entirely straightforward:

- **Case** $T = C \in \{1, Char, Int\}$:

$$\begin{array}{cl} & map\langle\!\langle C :: \star\rangle\!\rangle \in \mathcal{I}d\langle\star\rangle\ C \\ \equiv & \quad \{\text{ definition of } \mathcal{I}d \ \} \\ & map\langle\!\langle C :: \star\rangle\!\rangle = id :: C \to C \end{array}$$

$$
\begin{aligned}
\equiv\quad & \{\text{ definition of } map \}\\
& id = id :: C \to C\\
\equiv\quad & \{\text{ logic }\}\\
& true.
\end{aligned}
$$

- **Case** $T = (\bowtie) \in \{+, \times\}$:

$$
\begin{aligned}
& map\langle\!\langle(\bowtie) :: \star \to \star \to \star\rangle\!\rangle \in \mathcal{I}d\langle\star \to \star \to \star\rangle\ (\bowtie)\\
\equiv\quad & \{\text{ definition of } \mathcal{I}d \}\\
& \forall A :: \star\,.\,\forall B :: \star\,.\ map\langle\!\langle(\bowtie) :: \star \to \star \to \star\rangle\!\rangle\ A\ id\ B\ id = id :: A \bowtie B \to A \bowtie B\\
\equiv\quad & \{\text{ definition of } map \}\\
& \forall A :: \star\,.\,\forall B :: \star\,.\,(\bowtie)\ A\ id\ B\ id = id :: A \bowtie B \to A \bowtie B\\
\equiv\quad & \{\ (\bowtie) \text{ bifunctor }\}\\
& \forall A :: \star\,.\,\forall B :: \star\,.\ id = id :: A \bowtie B \to A \bowtie B\\
\equiv\quad & \{\text{ logic }\}\\
& true.
\end{aligned}
$$

The second law is shown analogously.

4.3.1 Soundness

Recall the basic idea of logical relations (Section 2.4.4). Say, we are given two models of the simply typed lambda calculus. Lemma 2.20, sometimes called the *Basic Lemma*, establishes that the meaning of a term in one model is logically related to its meaning in the other model.

We have said several times that the specialization of a generic value can be seen as an interpretation of the simply typed lambda calculus. Actually, the interpretation is a two-stage process: the specialization maps a type term to a value term, which is then interpreted in some fixed domain-theoretic model.

Consequently, there are two differences to the 'classical' notion of logical relation. (i) We do not relate elements in two different models but different elements (obtained via the specialization) in the same model, that is, for some fixed model the meaning of $poly_1\langle\!\langle T\rangle\!\rangle$ is logically related to the meaning of $poly_2\langle\!\langle T\rangle\!\rangle$. (ii) The type of $poly_1\langle\!\langle T\rangle\!\rangle$ and $poly_2\langle\!\langle T\rangle\!\rangle$ and consequently the type of their meanings depends on the type-index T. For that reason generic logical relations are parameterized by types (respectively, by the meaning of types).

For presenting the Basic Lemma of generic logical relations we will use the following 'semantic version' of *poly* (see also Section 3.1.3).

$$
\begin{aligned}
\mathbf{poly}\langle\!\langle C :: \mathfrak{C}\rangle\!\rangle\eta &= \mathbf{poly}_C\\
\mathbf{poly}\langle\!\langle A :: \mathfrak{A}\rangle\!\rangle\eta &= \eta(poly_A)\\
\mathbf{poly}\langle\!\langle (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta &= (\mathbf{poly}\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta, \mathbf{poly}\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta)\\
\mathbf{poly}\langle\!\langle Outl\ T :: \mathfrak{T}_1\rangle\!\rangle\eta &= \mathbf{outl}\ (\mathbf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta)\\
\mathbf{poly}\langle\!\langle Outr\ T :: \mathfrak{T}_2\rangle\!\rangle\eta &= \mathbf{outr}\ (\mathbf{poly}\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta)\\
\mathbf{poly}\langle\!\langle (\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta &= \boldsymbol{\lambda}\alpha_1 \ldots \alpha_n\,.\,\boldsymbol{\lambda}\varphi\,.\,\mathbf{poly}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta(A_1 := \alpha_1, \ldots, A_n := \alpha_n, poly_A := \varphi)\\
\mathbf{poly}\langle\!\langle T\ U :: \mathfrak{V}\rangle\!\rangle\eta &= (\mathbf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V}\rangle\!\rangle\eta)\ ([\![\lfloor U\rfloor_1]\!]\eta)\ \cdots\ ([\![\lfloor U\rfloor_n]\!]\eta)\ (\mathbf{poly}\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta)\\
\mathbf{poly}\langle\!\langle Fix\ T :: U\rangle\!\rangle\eta &= \mathbf{slfp}\ ([\![\lfloor T\rfloor_1]\!]\eta)\ \cdots\ ([\![\lfloor T\rfloor_n]\!]\eta)\ (\mathbf{poly}\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta)
\end{aligned}
$$

Here, **slfp** is the n-ary generalization of **slfp** introduced in Section 3.1.3.

In presenting logical relations we will restrict ourselves to the binary case. The extension to the n-ary case is entirely straightforward.

DEFINITION 4.2 Let Poly_1 and Poly_2 be two families of kind-indexed types $\mathsf{Poly}_i = (\mathsf{Poly}_i^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ such that $\mathsf{Poly}_i^{\mathfrak{T}} \in \mathbf{T}^{\mathfrak{T} \to \cdots \to \mathfrak{T} \to \star}$. A generic logical relation $\mathcal{R} = (\mathcal{R}^{\mathfrak{T}} \mid \mathfrak{T} \in \mathfrak{Kind})$ over Poly_1 and Poly_2 is a family of relations such that

- $\mathcal{R}^{\mathfrak{T}}\ \tau_1\ \ldots\ \tau_n \subseteq Dom\ (\mathsf{Poly}_1^{\mathfrak{T}}\ \tau_1\ \ldots\ \tau_n) \times Dom\ (\mathsf{Poly}_2^{\mathfrak{T}}\ \tau_1\ \ldots\ \tau_n)$ for all $\tau_1, \ldots, \tau_n \in \mathbf{T}^{\mathfrak{T}}$,
- $\mathcal{R}^{\mathfrak{T} \times \mathfrak{U}}$ is closed under pairing and projection:

$$\begin{aligned}(\varphi_1, \varphi_2) \in \mathcal{R}^{\mathfrak{T} \times \mathfrak{U}}\ \tau_1\ \ldots\ \tau_n \\ \equiv\ (\mathsf{outl}\ \varphi_1, \mathsf{outl}\ \varphi_2) \in \mathcal{R}^{\mathfrak{T}}\ (\mathsf{outl}\ \tau_1)\ \ldots\ (\mathsf{outl}\ \tau_n) \\ \cap\ (\mathsf{outr}\ \varphi_1, \mathsf{outr}\ \varphi_2) \in \mathcal{R}^{\mathfrak{U}}\ (\mathsf{outr}\ \tau_1)\ \ldots\ (\mathsf{outr}\ \tau_n),\end{aligned}$$

- $\mathcal{R}^{\mathfrak{T} \to \mathfrak{U}}$ is closed under application and abstraction:

$$\begin{aligned}&(\varphi_1, \varphi_2) \in \mathcal{R}^{\mathfrak{T} \to \mathfrak{U}}\ \tau_1\ \ldots\ \tau_n \\ &\quad \equiv \forall \alpha_1 \in \mathbf{T}^{\mathfrak{T}}\ . \ldots . \forall \alpha_n \in \mathbf{T}^{\mathfrak{T}}\ . \\ &\qquad \forall \delta_1 \in Dom\ (\mathsf{Poly}_1^{\mathfrak{T}}\ \alpha_1\ \ldots\ \alpha_n)\ . \\ &\qquad \forall \delta_2 \in Dom\ (\mathsf{Poly}_2^{\mathfrak{T}}\ \alpha_1\ \ldots\ \alpha_n)\ . \\ &\qquad\quad (\delta_1, \delta_2) \in \mathcal{R}^{\mathfrak{T}}\ \alpha_1\ \ldots\ \alpha_n \\ &\qquad\qquad \supset (\varphi_1\ \alpha_1\ \ldots\ \alpha_n\ \delta_1, \varphi_2\ \alpha_1\ \ldots\ \alpha_n\ \delta_2) \in \mathcal{R}^{\mathfrak{U}}\ (\tau_1\ \alpha_1)\ \ldots\ (\tau_n\ \alpha_n),\end{aligned}$$

- $\mathcal{R}^{\mathfrak{T}}$ is pointed, that is, $(\bot, \bot) \in \mathcal{R}^{\mathfrak{T}}\ \bot\ \cdots\ \bot$,
- $\mathcal{R}^{\mathfrak{T}}$ is chain-complete, that is, $S \subseteq \mathcal{P} \supset \bigsqcup S \in \mathcal{P}$ for every chain S where $\mathcal{P}(\alpha_1, \alpha_2; \tau_1, \ldots, \tau_n) \equiv (\alpha_1, \alpha_2) \in \mathcal{R}^{\mathfrak{T}}\ \tau_1\ \ldots\ \tau_n$. □

Without loss of generality we assume that the type arguments of $\mathcal{R}$ and Poly_i are the same (in general, the type arguments of Poly_i are a subset of the type arguments of $\mathcal{R}$).

REMARK 4.3 In models where types are interpreted as certain elements of a universal domain the notion of chain-completeness that is employed in Definition 4.2 coincides with the usual notion. Consider, for instance, the finitary projection model: the typed inequality $t_1 \sqsubseteq t_2 :: T$ is interpreted as $\tau\ [\![t_1]\!] \sqsubseteq \tau\ [\![t_2]\!]$ where $\tau = [\![T]\!]$ is a finitary projection. Consequently, a property that is built from equalities and inequalities using conjunction, disjunction and universal quantification always denotes a chain-complete relation. □

LEMMA 4.4 Let $\mathcal{R}$ be a generic logical relation over Poly_1 and Poly_2. Furthermore, let $\mathsf{poly}_1\langle\!\langle V :: \mathfrak{V}\rangle\!\rangle \in Dom\ (\mathsf{Poly}_1^{\mathfrak{V}}\ [\![T]\!]\ \cdots\ [\![T]\!])$ and $\mathsf{poly}_2\langle\!\langle V :: \mathfrak{V}\rangle\!\rangle \in Dom\ (\mathsf{Poly}_2^{\mathfrak{V}}\ [\![T]\!]\ \cdots\ [\![T]\!])$ be two generic function such that

$$(\mathsf{poly}_1\langle\!\langle C :: \mathfrak{C}\rangle\!\rangle, \mathsf{poly}_2\langle\!\langle C :: \mathfrak{C}\rangle\!\rangle) \in \mathcal{R}^{\mathfrak{C}}\ [\![C]\!]\ \cdots\ [\![C]\!]$$

for every type constant $C \in Const$. Let $V :: \mathfrak{V}$ be a monomorphic type term. If η_1, η_2, and $\varrho_1, \ldots, \varrho_n$ are environments such that $\eta_1(A_j) = \eta_2(A_j) = \varrho_j(A)$ and $(\eta_1(poly_A), \eta_2(poly_A)) \in \mathcal{R}^{\mathfrak{A}}\ (\varrho_1(A))\ \ldots\ (\varrho_n(A))$ for every type variable $A :: \mathfrak{A}$ free in $V :: \mathfrak{V}$, then

$$(\mathsf{poly}_1\langle\!\langle V :: \mathfrak{V}\rangle\!\rangle\eta_1, \mathsf{poly}_2\langle\!\langle V :: \mathfrak{V}\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{V}}\ ([\![V]\!]\varrho_1)\ \cdots\ ([\![V]\!]\varrho_n).$$

PROOF. We proceed by induction on the kinding derivation of $V :: \mathfrak{V}$.

- **Case** $V = C :: \mathfrak{C}$: the statement holds since $\mathcal{R}$ relates constants.
- **Case** $V = A :: \mathfrak{A}$: the statement holds since $\eta_1(\mathit{poly}_A)$ and $\eta_2(\mathit{poly}_A)$ are related.
- **Case** $V = (T_1, T_2) :: \mathfrak{T}_1 \times \mathfrak{T}_2$: by the induction hypothesis we have

$$(\mathbf{poly}_1\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta_1, \mathbf{poly}_2\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{T}_1}\ ([\![T_1]\!]\varrho_1)\ \cdots\ ([\![T_2]\!]\varrho_n)$$

and

$$(\mathbf{poly}_1\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta_1, \mathbf{poly}_2\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{T}_2}\ ([\![T_2]\!]\varrho_1)\ \cdots\ ([\![T_2]\!]\varrho_n),$$

which immediately implies

$$\begin{aligned}&((\mathbf{poly}_1\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta_1, \mathbf{poly}_1\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta_1), (\mathbf{poly}_2\langle\!\langle T_1 :: \mathfrak{T}_1\rangle\!\rangle\eta_2, \mathbf{poly}_2\langle\!\langle T_2 :: \mathfrak{T}_2\rangle\!\rangle\eta_2))\\ &\quad \in \mathcal{R}^{\mathfrak{T}_1\times\mathfrak{T}_2}\ ([\![T_1]\!]\varrho_1, [\![T_2]\!]\varrho_1)\ \ldots\ ([\![T_1]\!]\varrho_n, [\![T_2]\!]\varrho_n).\end{aligned}$$

- **Case** $V = \mathit{Outl}\ T :: \mathfrak{T}_1$: by the induction hypothesis we have

$$(\mathbf{poly}_1\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta_1, \mathbf{poly}_2\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle) \in \mathcal{R}^{\mathfrak{V}}\ ([\![T]\!]\varrho_1)\ \cdots\ ([\![T]\!]\varrho_n),$$

which immediately implies

$$\begin{aligned}&(\mathbf{outl}\ (\mathbf{poly}_1\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle\eta_1), \mathbf{outl}\ (\mathbf{poly}_2\langle\!\langle T :: \mathfrak{T}_1 \times \mathfrak{T}_2\rangle\!\rangle))\\ &\quad \in \mathcal{R}^{\mathfrak{T}}\ (\mathbf{outl}\ ([\![T]\!]\varrho_1))\ \ldots\ (\mathbf{outl}\ ([\![T]\!]\varrho_n)).\end{aligned}$$

- **Case** $V = \mathit{Outr}\ T :: \mathfrak{T}_2$: analogous.
- **Case** $V = (\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}$: We have to show that

$$\begin{aligned}&(\mathbf{poly}_1\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta_1, \mathbf{poly}_2\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{S}\to\mathfrak{T}}\ ([\![\Lambda A\,.\,T]\!]\varrho_1)\ldots([\![\Lambda A\,.\,T]\!]\varrho_n)\\ &\quad \equiv \forall\alpha_1 \in \mathbf{T}^{\mathfrak{S}}\,.\,\ldots\,.\forall\alpha_n \in \mathbf{T}^{\mathfrak{S}}\,.\\ &\qquad\quad \forall\delta_1 \in \mathit{Dom}\ (\mathbf{Poly}_1^{\mathfrak{S}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\ &\qquad\quad \forall\delta_2 \in \mathit{Dom}\ (\mathbf{Poly}_2^{\mathfrak{S}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\ &\qquad\qquad (\delta_1, \delta_2) \in \mathcal{R}^{\mathfrak{S}}\ \alpha_1\ \ldots\ \alpha_n\\ &\qquad\qquad\quad \supset (\mathbf{poly}_1\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta_1\ \alpha_1\ \ldots\ \alpha_n\ \delta_1,\\ &\qquad\qquad\qquad \mathbf{poly}_2\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta_2\ \alpha_1\ \ldots\ \alpha_n\ \delta_2)\\ &\qquad\qquad\qquad\quad \in \mathcal{R}^{\mathfrak{T}}\ ([\![\Lambda A\,.\,T]\!]\varrho_1\ \alpha_1)\ \ldots\ ([\![\Lambda A\,.\,T]\!]\varrho_n\ \alpha_n),\end{aligned}$$

Assume that $(\delta_1, \delta_2) \in \mathcal{R}^{\mathfrak{S}}\ \alpha_1\ \ldots\ \alpha_n$. Since the modified environments $\eta_1(A_1{:=}\alpha_1, \ldots, A_n{:=}\alpha_n, \mathit{poly}_A{:=}\delta_1)$, $\eta_2(A_1{:=}\alpha_1, \ldots, A_n{:=}\alpha_n, \mathit{poly}_A{:=}\delta_2)$, and $\varrho_1(A := \alpha_1)$, …, $\varrho_n(A := \alpha_n)$ are related, we can invoke the induction hypothesis to obtain

$$\begin{aligned}&(\mathbf{poly}_1\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta_1(A_1 := \alpha_1, \ldots, A_n := \alpha_n, \mathit{poly}_A := \delta_1),\\ &\ \mathbf{poly}_2\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta_2(A_1 := \alpha_1, \ldots, A_n := \alpha_n, \mathit{poly}_A := \delta_2))\\ &\qquad \in \mathcal{R}^{\mathfrak{V}}\ ([\![T]\!]\varrho_1(A := \alpha_1))\ \cdots\ ([\![T]\!]\varrho_n(A := \alpha_n)).\end{aligned}$$

Now, since

$$\mathsf{poly}_i\langle\!\langle(\Lambda A\,.\,T) :: \mathfrak{S} \to \mathfrak{T}\rangle\!\rangle\eta_i\ \alpha_1\ \ldots\ \alpha_n\ \delta_i = \mathsf{poly}_i\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\eta_i\ (A_1 := \alpha_1, \ldots, A_n := \alpha_n, \mathit{poly}_A := \delta_i)$$

and furthermore $[\![\Lambda A\,.\,T]\!]\varrho_j\ \alpha_j = [\![T]\!]\varrho_j(A := \alpha_j)$ the proposition follows.

- **Case** $V = (T\ U) :: \mathfrak{V}$: by the induction hypothesis we have

$$\begin{aligned}
&(\mathsf{poly}_1\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V}\rangle\!\rangle\eta_1, \mathsf{poly}_2\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V}\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{U}\to\mathfrak{V}}\ ([\![T]\!]\varrho_1)\ \cdots\ ([\![T]\!]\varrho_n)\\
&\quad\equiv \forall\alpha_1 \in \mathbf{T}^{\mathfrak{U}}\,.\ \ldots\,.\,\forall\alpha_n \in \mathbf{T}^{\mathfrak{U}}\,.\\
&\qquad\qquad \forall\delta_1 \in \mathit{Dom}\ (\mathsf{Poly}_1^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\
&\qquad\qquad \forall\delta_2 \in \mathit{Dom}\ (\mathsf{Poly}_2^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\
&\qquad\qquad\quad (\delta_1, \delta_2) \in \mathcal{R}^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n\\
&\qquad\qquad\qquad \supset (\varphi_1\ \alpha_1\ \ldots\ \alpha_n\ \delta_1, \varphi_2\ \alpha_1\ \ldots\ \alpha_n\ \delta_2) \in \mathcal{R}^{\mathfrak{V}}\ ([\![T]\!]\varrho_1\ \alpha_1)\ \ldots\ ([\![T]\!]\varrho_n\ \alpha_n)
\end{aligned}$$

and

$$(\mathsf{poly}_1\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta_1, \mathsf{poly}_2\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{U}}\ ([\![U]\!]\varrho_1)\ \cdots\ ([\![U]\!]\varrho_n).$$

Setting $\alpha_j = [\![\lfloor U\rfloor_j]\!]\eta_1 = [\![\lfloor U\rfloor_j]\!]\eta_2$ and $\delta_i = \mathsf{poly}_i\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta_i$ and since $[\![\lfloor U\rfloor_j]\!]\eta_1 = [\![\lfloor U\rfloor_j]\!]\eta_2 = [\![U]\!]\varrho_j$, we obtain

$$\begin{aligned}
&((\mathsf{poly}_1\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V}\rangle\!\rangle\eta_1)\ ([\![\lfloor U\rfloor_1]\!]\eta_1)\ \cdots\ ([\![\lfloor U\rfloor_n]\!]\eta_1)\ (\mathsf{poly}\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta_1),\\
&\ (\mathsf{poly}_2\langle\!\langle T :: \mathfrak{U} \to \mathfrak{V}\rangle\!\rangle\eta_2)\ ([\![\lfloor U\rfloor_1]\!]\eta_2)\ \cdots\ ([\![\lfloor U\rfloor_n]\!]\eta_2)\ (\mathsf{poly}\langle\!\langle U :: \mathfrak{U}\rangle\!\rangle\eta_2))\\
&\qquad \in \mathcal{R}^{\mathfrak{V}}\ (([\![T]\!]\varrho_1)\ ([\![U]\!]\varrho_1))\ \ldots\ (([\![T]\!]\varrho_n)\ ([\![U]\!]\varrho_n)).
\end{aligned}$$

- **Case** $V = \mathit{Fix}\ T :: \mathfrak{U}$: by the induction hypothesis we have

$$\begin{aligned}
&(\mathsf{poly}_1\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta_1, \mathsf{poly}_2\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta_2) \in \mathcal{R}^{\mathfrak{U}\to\mathfrak{U}}\ ([\![T]\!]\varrho_1)\ \cdots\ ([\![T]\!]\varrho_n)\\
&\quad\equiv \forall\alpha_1 \in \mathbf{T}^{\mathfrak{U}}\,.\ \ldots\,.\,\forall\alpha_n \in \mathbf{T}^{\mathfrak{U}}\,.\\
&\qquad\qquad \forall\delta_1 \in \mathit{Dom}\ (\mathsf{Poly}_1^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\
&\qquad\qquad \forall\delta_2 \in \mathit{Dom}\ (\mathsf{Poly}_2^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n)\,.\\
&\qquad\qquad\quad (\delta_1, \delta_2) \in \mathcal{R}^{\mathfrak{U}}\ \alpha_1\ \ldots\ \alpha_n\\
&\qquad\qquad\qquad \supset (\mathsf{poly}_1\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta_1\ \alpha_1\ \ldots\ \alpha_n\ \delta_1, \mathsf{poly}_2\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta_2\ \alpha_1\ \ldots\ \alpha_n\ \delta_2)\\
&\qquad\qquad\qquad\qquad \in \mathcal{R}^{\mathfrak{U}}\ ([\![T]\!]\varrho_1\ \alpha_1)\ \ldots\ ([\![T]\!]\varrho_n\ \alpha_n),
\end{aligned}$$

Define

$$\begin{array}{lllllll}
\alpha_1^0 & = & \perp & \qquad & \alpha_1^{k+1} & = & [\![T]\!]\varrho_1\ \alpha_1^k\\
\ldots & & & & \ldots & & \\
\alpha_n^0 & = & \perp & & \alpha_n^{k+1} & = & [\![T]\!]\varrho_n\ \alpha_n^k
\end{array}$$

and

$$\begin{array}{lll}
\delta_i^0 & = & \perp\\
\delta_i^{k+1} & = & \mathsf{poly}_i\langle\!\langle T :: \mathfrak{U} \to \mathfrak{U}\rangle\!\rangle\eta_i\ \alpha_1^k\ \ldots\ \alpha_n^k\ \delta_i^k.
\end{array}$$

Using the induction hypothesis and the fact that $\mathcal{R}^{\mathfrak{U}}$ is pointed we can show

$$(\delta_1^k, \delta_2^k) \in \mathcal{R}^{\mathfrak{U}}\ \alpha_1^k\ \ldots\ \alpha_n^k,$$

for all $k \in \mathbb{N}$. Because $\mathcal{R}^{\mathfrak{U}}$ is furthermore chain-complete, we have

$$(\bigsqcup\{\delta_1^k \mid k \in \mathbb{N}\}, \bigsqcup\{\delta_2^k \mid k \in \mathbb{N}\}) \in \mathcal{R}^{\mathfrak{U}}\ (\bigsqcup\{\alpha_1^k \mid k \in \mathbb{N}\})\ \ldots\ (\bigsqcup\{\alpha_n^k \mid k \in \mathbb{N}\}).$$

Now, since

$$\bigsqcup\{\alpha_j^k \mid k \in \mathbb{N}\} = \textsf{lfp}\ (\llbracket T \rrbracket \varrho_j) = \textsf{lfp}\ (\llbracket \lfloor T \rfloor_j \rrbracket \eta_1) = \textsf{lfp}\ (\llbracket \lfloor T \rfloor_j \rrbracket \eta_2)$$

and

$$\begin{aligned}
& \bigsqcup\{\delta_i^k \mid k \in \mathbb{N}\} \\
= {} & \quad \{ \text{ definition of } \textsf{slfp} \} \\
& \textsf{slfp}\ (\llbracket T \rrbracket \varrho_1)\ \ldots\ (\llbracket T \rrbracket \varrho_n)\ (\textsf{poly}_i \langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle \eta_i) \\
= {} & \quad \{ \llbracket T \rrbracket \varrho_j = \llbracket \lfloor T \rfloor_j \rrbracket \eta_1 = \llbracket \lfloor T \rfloor_j \rrbracket \eta_2 \} \\
& \textsf{slfp}\ (\llbracket \lfloor T \rfloor_1 \rrbracket \eta_i)\ \cdots\ (\llbracket \lfloor T \rfloor_n \rrbracket \eta_i)\ (\textsf{poly}_i \langle\!\langle T :: \mathfrak{U} \to \mathfrak{U} \rangle\!\rangle \eta_i)
\end{aligned}$$

the proposition follows. □

4.3.2 Examples

Let us illustrate the proof technique by means of some further examples.

A fusion law for ***count*** Many generic properties take the form of *fusion laws*, which show how to fuse a composition of two functions into a single function. As an example, let us formulate a fusion law for the generic function *count* defined in Section 3.3.3. Let $h :: \mathit{Int} \to \mathit{Int}$ and define $\mathcal{F}\mathit{use}_h$ by

$$\begin{aligned}
\mathcal{F}\mathit{use}_h \langle \mathfrak{T} \rangle\ T & \subseteq \mathit{Count}\langle \mathfrak{T} \rangle\ T \times \mathit{Count}\langle \mathfrak{T} \rangle\ T \\
(c, c') \in \mathcal{F}\mathit{use}_h \langle \star \rangle\ T & \equiv h \cdot c = c' :: T \to \mathit{Int}.
\end{aligned}$$

We seek conditions so that

$$(\mathit{count}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle, \mathit{count}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle) \in \mathcal{F}\mathit{use}_h \langle \mathfrak{T} \rangle\ T$$

holds. First of all, $\mathcal{F}\mathit{use}_h$ is pointed iff h is strict: $(\bot, \bot) \in \mathcal{F}\mathit{use}_h \langle \star \rangle\ 0 \equiv h \cdot \bot = \bot$.

- **Case** $T = C \in \{1, \mathit{Char}, \mathit{Int}\}$:

$$\begin{aligned}
& (\mathit{count}\langle\!\langle C \rangle\!\rangle, \mathit{count}\langle\!\langle C \rangle\!\rangle) \in \mathcal{F}\mathit{use}_h \langle \star \rangle\ C \\
\equiv {} & \quad \{ \text{ definition of } \mathcal{F}\mathit{use}_h \} \\
& h \cdot \mathit{count}\langle\!\langle C \rangle\!\rangle = \mathit{count}\langle\!\langle C \rangle\!\rangle \\
\equiv {} & \quad \{ \text{ definition of } \mathit{count} \} \\
& h \cdot k\ 0 = k\ 0.
\end{aligned}$$

  Consequently, we must postulate $h\ 0 = 0$.

- **Case** $T = (+)$:

$$\begin{aligned}
& (\mathit{count}\langle\!\langle + \rangle\!\rangle, \mathit{count}\langle\!\langle + \rangle\!\rangle) \in \mathcal{F}\mathit{use}_h \langle \star \to \star \to \star \rangle\ (+) \\
\equiv {} & \quad \{ \text{ definition of } \mathcal{F}\mathit{use}_h \} \\
& h \cdot \mathit{count}\langle\!\langle + \rangle\!\rangle\ c_1\ c_2 = \mathit{count}\langle\!\langle + \rangle\!\rangle\ (h \cdot c_1)\ (h \cdot c_2) \\
\equiv {} & \quad \{ \text{ definition of } \mathit{count} \} \\
& h \cdot (c_1 \triangledown c_2) = (h \cdot c_1) \triangledown (h \cdot c_2) \\
\equiv {} & \quad \{ \triangledown\text{-fusion law: } h \cdot (f \triangledown g) = (h \cdot f) \triangledown (h \cdot g) \} \\
& \mathit{true}.
\end{aligned}$$

So, this case comes for free.

- **Case** $T = (\times)$:

$$
\begin{aligned}
& (count\langle\!\langle \times \rangle\!\rangle, count\langle\!\langle \times \rangle\!\rangle) \in \mathcal{F}use_h\langle \star \to \star \to \star \rangle\ (\times) \\
\equiv \quad & \quad \{ \text{ definition of } \mathcal{F}use_h \ \} \\
& h \cdot count\langle\!\langle \times \rangle\!\rangle\ c_1\ c_2 = count\langle\!\langle \times \rangle\!\rangle\ (h \cdot c_1)\ (h \cdot c_2) \\
\equiv \quad & \quad \{ \text{ definition of } count \ \} \\
& h \cdot plus \cdot (c_1 \times c_2) = plus \cdot (h \cdot c_1) \times (h \cdot c_2) \\
\subset \quad & \quad \{ \ (\times) \text{ bifunctor } \} \\
& h \cdot plus = plus \cdot (h \times h).
\end{aligned}
$$

Consequently, we must postulate $h\ (i + j) = h\ i + h\ j$.

To summarize, we have derived the following fusion law for *count*:

$$
\begin{aligned}
& h\ \bot = \bot \\
\cap \quad & h\ 0 = 0 \\
\cap \quad & h\ (i + j) = h\ i + h\ j \\
\supset \quad & (count\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle, count\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle) \in \mathcal{F}use_h\langle \mathfrak{T} \rangle\ T.
\end{aligned}
$$

As an application of the law here is a more compact proof of $size\langle A \rangle = k\ a \supset size\langle T \cdot A \rangle = times\ a \cdot size\langle T \rangle$, see Section 4.1.2:

$$
\begin{aligned}
& size\langle T \cdot A \rangle \\
= \quad & \quad \{ \text{ definition of } size \ \} \\
& count\langle\!\langle T \cdot A \rangle\!\rangle\ (k\ 1) \\
\equiv \quad & \quad \{ \text{ definition of } count \ \} \\
& count\langle\!\langle T \rangle\!\rangle\ (count\langle\!\langle A \rangle\!\rangle\ (k\ 1)) \\
= \quad & \quad \{ \text{ definition of } size \ \} \\
& count\langle\!\langle T \rangle\!\rangle\ (size\langle A \rangle) \\
= \quad & \quad \{ \text{ assumption: } size\langle A \rangle = k\ a \ \} \\
& count\langle\!\langle T \rangle\!\rangle\ (k\ a) \\
= \quad & \quad \{ \ count\text{-fusion: } h = times\ a \ \} \qquad (\dagger) \\
& times\ a \cdot count\langle\!\langle T \rangle\!\rangle\ (k\ 1) \\
= \quad & \quad \{ \text{ definition of } size \ \} \\
& times\ a \cdot size\langle T \rangle.
\end{aligned}
$$

For ($\dagger$) we have to show that *times a* is strict, that is, $a \times \bot = \bot$, that $times\ a \cdot k\ 0 = k\ 0$, that is, $a \times 0 = 0$ and finally that $times\ a \cdot plus = plus \cdot (times\ a \times times\ a)$, that is, $a \times (b + c) = (a \times b) + (a \times c)$. All of these conditions hold.

Coping with $\bot$ Reconsider the definition of *count*. One is tempted to assume that $count\langle\!\langle T :: \star \rangle\!\rangle\ t = 0$ for all types T of kind $\star$. However, in a non-strict language such as Haskell this law only holds provided t is finite and fully defined. This example shows how to deal with this restriction in a systematic way. To this end we introduce a function that fully evaluates its argument.

$$
\begin{array}{lcl}
\mathit{Force}\langle\mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \star \\
\mathit{Force}\langle\star\rangle\ T & = & T \to () \\
\mathit{Force}\langle\mathfrak{A} \times \mathfrak{B}\rangle\ T & = & \mathit{Force}\langle\mathfrak{A}\rangle\ (\mathit{Outl}\ T) \times \mathit{Force}\langle\mathfrak{B}\rangle\ (\mathit{Outr}\ T) \\
\mathit{Force}\langle\mathfrak{A} \to \mathfrak{B}\rangle\ T & = & \forall A\,.\,\mathit{Force}\langle\mathfrak{A}\rangle\ A \to \mathit{Force}\langle\mathfrak{B}\rangle\ (T\ A) \\
\mathit{force}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & \mathit{Force}\langle\mathfrak{T}\rangle\ T \\
\mathit{force}\langle\!\langle 1\rangle\!\rangle\ u & = & u \text{ `}\mathit{seq}\text{` } () \\
\mathit{force}\langle\!\langle \mathit{Char}\rangle\!\rangle\ c & = & c \text{ `}\mathit{seq}\text{` } () \\
\mathit{force}\langle\!\langle \mathit{Int}\rangle\!\rangle\ i & = & i \text{ `}\mathit{seq}\text{` } () \\
\mathit{force}\langle\!\langle +\rangle\!\rangle\ \mathit{fA}\ \mathit{fB}\ (\mathit{inl}\ a) & = & \mathit{fA}\ a \\
\mathit{force}\langle\!\langle +\rangle\!\rangle\ \mathit{fA}\ \mathit{fB}\ (\mathit{inr}\ b) & = & \mathit{fB}\ b \\
\mathit{force}\langle\!\langle \times\rangle\!\rangle\ \mathit{fA}\ \mathit{fB}\ (a, b) & = & \mathit{fA}\ a \text{ `}\mathit{seq}\text{` } \mathit{fB}\ b
\end{array}
$$

The Haskell function $\mathit{seq} :: \forall A\ B\,.\,A \to B \to B$ evaluates its first argument to weak head-normal form and returns its second argument.

Using $\mathit{force}\langle T\rangle$ we can state the law concerning *count* more precisely

$$\mathit{force}\langle\!\langle T :: \star\rangle\!\rangle\ t \neq \bot\ \supset\ \mathit{count}\langle\!\langle T :: \star\rangle\!\rangle\ t = 0.$$

The precondition $\mathit{force}\langle\!\langle T :: \star\rangle\!\rangle\ t \neq \bot$ formalizes that t is finite and fully defined. Note that the property is chain-complete, since we can rewrite it into the form

$$\mathit{force}\langle\!\langle T :: \star\rangle\!\rangle\ t = \bot\ \cup\ \mathit{count}\langle\!\langle T :: \star\rangle\!\rangle\ t = 0,$$

which is chain-complete. Phrasing the property as a logical relation

$$
\begin{array}{lcl}
\mathcal{C}\mathit{onst}\langle\mathfrak{T}\rangle\ T & \subseteq & \mathit{Force}\langle\mathfrak{T}\rangle\ T \times \mathit{Count}\langle\mathfrak{T}\rangle\ T \\
(e, c) \in \mathcal{C}\mathit{onst}\langle\star\rangle\ T & \equiv & \forall t :: T\,.\,e\ t \neq \bot\ \supset\ c\ t = 0,
\end{array}
$$

we have to show that

$$(\mathit{force}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, \mathit{count}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle) \in \mathcal{C}\mathit{onst}\langle\mathfrak{T}\rangle\ T.$$

The proof is as follows:

- **Case** $T = C \in \{1, \mathit{Char}, \mathit{Int}\}$: We have to show that

  $$
  \begin{array}{l}
  (\mathit{force}\langle\!\langle C\rangle\!\rangle, \mathit{count}\langle\!\langle C\rangle\!\rangle) \in \mathcal{C}\mathit{onst}\langle\star\rangle\ C \\
  \quad \equiv\ \forall c \in C\,.\,\mathit{force}\langle\!\langle C\rangle\!\rangle\ c \neq \bot\ \supset\ \mathit{count}\langle\!\langle C\rangle\!\rangle\ c = 0,
  \end{array}
  $$

  which holds since $\mathit{count}\langle\!\langle C\rangle\!\rangle\ c = 0$.

- **Case** $T = (+)$: We have to show that

$$
\begin{array}{l}
(\mathit{force}\langle\!\langle +\rangle\!\rangle, \mathit{count}\langle\!\langle +\rangle\!\rangle) \in \mathcal{C}\mathit{onst}\langle\star \to \star \to \star\rangle\ (+) \\
\quad \equiv\ (\forall a \in A\,.\,\mathit{fA}\ a \neq \bot\ \supset\ \mathit{cA}\ a = 0) \\
\qquad\qquad \supset\ (\forall b \in B\,.\,\mathit{fB}\ b \neq \bot\ \supset\ \mathit{cB}\ b = 0) \\
\qquad\qquad\qquad \supset\ (\forall s \in A + B\,.\,\mathit{force}\langle\!\langle +\rangle\!\rangle\ \mathit{fA}\ \mathit{fB}\ s \neq \bot\ \supset\ \mathit{count}\langle\!\langle +\rangle\!\rangle\ \mathit{cA}\ \mathit{cB}\ s = 0)
\end{array}
$$

  Now, $\mathit{force}\langle\!\langle +\rangle\!\rangle\ \mathit{fA}\ \mathit{fB}\ s \neq \bot$ implies $s \neq \bot$, so we only have to consider $s = \mathit{inl}\ a$ and $s = \mathit{inr}\ b$. If $s = \mathit{inl}\ a$, we furthermore know that $\mathit{fA}\ a \neq \bot$ and similarly for $s = \mathit{inr}\ b$.

Case $s = inl\ a$:

$$
\begin{array}{ll}
 & count\langle\!\langle + \rangle\!\rangle\ cA\ cB\ (inl\ a) \\
= & \quad \{ \text{definition of } count \} \\
 & cA\ a \\
= & \quad \{ \text{assumption } fA\ a \neq \bot \supset cA\ a = 0 \text{ and } fA\ a \neq \bot \} \\
 & 0.
\end{array}
$$

Case $s = inr\ b$: analogous.

- **Case** $T = (\times)$: We have to show that

$$
\begin{array}{l}
(force\langle\!\langle \times \rangle\!\rangle, count\langle\!\langle \times \rangle\!\rangle) \in \mathcal{C}onst\langle \star \to \star \to \star \rangle\ (\times) \\
\quad \equiv (\forall a \in A \,.\, fA\ a \neq \bot \supset cA\ a = 0) \\
\qquad\quad \supset (\forall b \in B \,.\, fB\ b \neq \bot \supset cB\ b = 0) \\
\qquad\qquad\quad \supset (\forall p \in A \times B \,.\, force\langle\!\langle \times \rangle\!\rangle\ fA\ fB\ p \neq \bot \supset count\langle\!\langle \times \rangle\!\rangle\ cA\ cB\ p = 0)
\end{array}
$$

Again, $force\langle\!\langle \times \rangle\!\rangle\ fA\ fB\ p \neq \bot$ implies $p \neq \bot$, so we only have to consider $p = (a, b)$. Furthermore, we know that both $fA\ a \neq \bot$ and $fB\ b \neq \bot$.

$$
\begin{array}{ll}
 & count\langle\!\langle \times \rangle\!\rangle\ cA\ cB\ (a, b) \\
= & \quad \{ \text{definition of } count \} \\
 & cA\ a + cB\ b \\
= & \quad \{ \text{assumptions and } fA\ a \neq \bot \cap fB\ b \neq \bot \} \\
 & 0 + 0 \\
= & \quad \{ \text{arithmetic} \} \\
 & 0.
\end{array}
$$

Chapter 5

Examples

This chapter presents further examples of generic values and associated generic proofs. Among other things, we study comparison functions (Section 5.1), mapping functions (Section 5.2), zipping functions (Section 5.3) and reductions (Section 5.4). Section 5.5 introduces an interesting extension of the theory developed in the previous chapters: type-indexed types and kind-indexed kinds. We use these techniques to implement dictionaries (Section 5.5) and memo tables (Section 5.6) in a generic way.

5.1 Comparison functions

In Section 3.1 we have introduced a generic version of the equality function. Varying the definition of *equal* slightly we can also realize Haskell's *compare* function, which determines the precise ordering of two elements.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Ordering} & = & \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT} \\
\mathit{compare}\langle T :: \star\rangle & :: & T \to T \to \mathit{Ordering} \\
\mathit{compare}\langle 1\rangle\ ()\ () & = & \mathit{EQ} \\
\mathit{compare}\langle \mathit{Char}\rangle\ c_1\ c_2 & = & \mathit{compareChar}\ c_1\ c_2 \\
\mathit{compare}\langle \mathit{Int}\rangle\ i_1\ i_2 & = & \mathit{compareInt}\ i_1\ i_2 \\
\mathit{compare}\langle A + B\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inl}\ a_2) & = & \mathit{compare}\langle A\rangle\ a_1\ a_2 \\
\mathit{compare}\langle A + B\rangle\ (\mathit{inl}\ a_1)\ (\mathit{inr}\ b_2) & = & \mathit{LT} \\
\mathit{compare}\langle A + B\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inl}\ a_2) & = & \mathit{GT} \\
\mathit{compare}\langle A + B\rangle\ (\mathit{inr}\ b_1)\ (\mathit{inr}\ b_2) & = & \mathit{compare}\langle B\rangle\ b_1\ b_2 \\
\mathit{compare}\langle A \times B\rangle\ (a_1, b_1)\ (a_2, b_2) & = & \mathit{compare}\langle A\rangle\ a_1\ a_2\ \text{`}\mathit{lexord}\text{`}\ \mathit{compare}\langle B\rangle\ b_1\ b_2
\end{array}
$$

The helper function *lexord* used in the last equation implements the lexicographic product of two orderings.

$$
\begin{array}{lcl}
\mathit{lexord} & :: & \mathit{Ordering} \to \mathit{Ordering} \to \mathit{Ordering} \\
\mathit{lexord}\ \mathit{LT}\ \mathit{ord} & = & \mathit{LT} \\
\mathit{lexord}\ \mathit{EQ}\ \mathit{ord} & = & \mathit{ord} \\
\mathit{lexord}\ \mathit{GT}\ \mathit{ord} & = & \mathit{GT}
\end{array}
$$

Note that *equal* and *compare* are related by

$$\mathit{equal}\langle T\rangle\ t_1\ t_2 = \mathit{compare}\langle T\rangle\ t_1\ t_2 == \mathit{EQ}.$$

The MPC-style version of *compare* has type $\mathit{Compare}\langle\mathfrak{T}\rangle\ T\ T$ where *Compare* is given by

$$
\begin{array}{lcl}
\mathit{Compare}\langle \mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \mathfrak{T} \to \star \\
\mathit{Compare}\langle \star\rangle\ T_1\ T_2 & = & T_1 \to T_2 \to \mathit{Ordering} \\
\mathit{Compare}\langle \mathfrak{A} \times \mathfrak{B}\rangle\ T_1\ T_2 & = & \mathit{Compare}\langle\mathfrak{A}\rangle\ (\mathit{Outl}\ T_1)\ (\mathit{Outl}\ T_2) \\
 & & \quad \times\ \mathit{Compare}\langle\mathfrak{B}\rangle\ (\mathit{Outr}\ T_1)\ (\mathit{Outr}\ T_2) \\
\mathit{Compare}\langle \mathfrak{A} \to \mathfrak{B}\rangle\ T_1\ T_2 & = & \forall A_1\ A_2\ .\ \mathit{Compare}\langle\mathfrak{A}\rangle\ A_1\ A_2 \\
 & & \qquad \to \mathit{Compare}\langle\mathfrak{B}\rangle\ (T_1\ A_1)\ (T_2\ A_2).
\end{array}
$$

Note that *Compare* corresponds to *PEqual*, the second, more general type of *equal*.

5.2 Mapping functions

In this section we take a look at two variations of mapping functions: embedding-projection maps (Section 5.2.1) and monadic maps (Section 5.2.2). Embedding-projection maps are useful for programming 'representation changers'; we will make intensive use of these maps in Chapter 6 when we discuss the implementation of Generic Haskell. Monadic maps can be used to thread a monad through a data structure; Section 5.2.2 contains an application along these lines.

5.2.1 Embedding-projection maps

Most of the generic functions cannot sensibly be defined for the function space. For instance, *map* cannot be defined for functional types since $(\rightarrow)$ is contravariant in its first argument:

$$
\begin{array}{lcl}
(\rightarrow) & :: & \forall A_1\ A_2\,.\,(A_2 \rightarrow A_1) \rightarrow \forall B_1\ B_2\,.\,(B_1 \rightarrow B_2) \rightarrow ((A_1 \rightarrow B_1) \rightarrow (A_2 \rightarrow B_2)) \\
(f \rightarrow g)\ h & = & g \cdot h \cdot f.
\end{array}
$$

Drawing from the theory of embeddings and projections (Gierz, Hofmann, Keimel, Lawson, Mislove, and Scott 1980) we can remedy the situation as follows. The central idea is to supply a pair of functions, *from* and *to*, where *to* is the left-inverse of *from*, that is, $to \cdot from = id$. If the functions additionally satisfy $from \cdot to \sqsubseteq id$, then they are called an *embedding-projection pair*. We use the following data type to represent embedding-projection pairs.

$$
\begin{array}{lcl}
\mathbf{data}\ EP\ A_1\ A_2 & = & ep\{from :: A_1 \rightarrow A_2, to :: A_2 \rightarrow A_1\} \\
idE & :: & \forall A\,.\,EP\ A\ A \\
idE & = & ep\{from = id, to = id\} \\
(-)^{op} & :: & \forall A_1\ A_2\,.\,EP\ A_1\ A_2 \rightarrow EP\ A_2\ A_1 \\
f^{op} & = & ep\{from = to\ f, to = from\ f\} \\
(\circ) & :: & \forall A\ B\ C\,.\,EP\ B\ C \rightarrow EP\ A\ B \rightarrow EP\ A\ C \\
f \circ g & = & ep\{from = from\ f \cdot from\ g, to = to\ g \cdot to\ f\}
\end{array}
$$

Here, *idE* is the identity embedding-projection pair and '$\circ$' shows how to compose two embedding-projection pairs (note that the composition is reversed for the projection). In fact, *idE* and '$\circ$' give rise to the category $\mathcal{C}po^e$, the category of complete partial orders and embedding-projection pairs. Note that m is an embedding-projection pair iff

$$
to\ m \cdot from\ m = id \cap from\ m \cdot to\ m \sqsubseteq id.
$$

POPL-style definition Given the definitions above we can define a variant of *map*, which additionally works for the function space constructor:

$$
\begin{array}{lcl}
mapE\langle T :: \star \rightarrow \star\rangle & :: & \forall A_1\ A_2\,.\,EP\ A_1\ A_2 \rightarrow (T\ A_1 \rightarrow T\ A_2) \\
mapE\langle Id\rangle\ m & = & from\ m \\
mapE\langle \underline{1}\rangle\ m & = & id \\
mapE\langle \underline{Char}\rangle\ m & = & id \\
mapE\langle \underline{Int}\rangle\ m & = & id \\
mapE\langle F \mathbin{\underline{+}} G\rangle\ m & = & mapE\langle F\rangle\ m + mapE\langle G\rangle\ m \\
mapE\langle F \mathbin{\underline{\times}} G\rangle\ m & = & mapE\langle F\rangle\ m \times mapE\langle G\rangle\ m \\
mapE\langle F \mathbin{\underline{\rightarrow}} G\rangle\ m & = & mapE\langle F\rangle\ m^{op} \rightarrow mapE\langle G\rangle\ m.
\end{array}
$$

Now, if F is a covariant functor (in $\mathcal{C}po$), we can define its mapping function in terms of $mapE$:

$$\begin{array}{lcl} map\langle F :: \star \to \star\rangle & :: & \forall A_1\ A_2 \,.\, (A_1 \to A_2) \to (T\ A_1 \to T\ A_2) \\ map\langle F\rangle\ m & = & mapE\langle F\rangle\ (ep\{from = m, to = \bot\}). \end{array}$$

Note that this definition is more general than the original definition of map since F may involve functional types as in $F = \Lambda A \,.\, S \to A \times S$. However, if F is not covariant, then we get a run-time error.

On the other hand, if F is a contravariant functor, we can define its mapping function also in terms of $mapE$:

$$\begin{array}{lcl} comap\langle F :: \star \to \star\rangle & :: & \forall A_1\ A_2 \,.\, (A_2 \to A_1) \to (T\ A_1 \to T\ A_2) \\ comap\langle F\rangle\ m & = & mapE\langle F\rangle\ (ep\{from = \bot, to = m\}). \end{array}$$

Finally, if F is neither covariant nor contravariant, then we can define a mapping function with a restricted type:

$$\begin{array}{lcl} endomap\langle F :: \star \to \star\rangle & :: & \forall A \,.\, (A \to A) \to (T\ A \to T\ A) \\ endomap\langle F\rangle\ m & = & mapE\langle F\rangle\ (ep\{from = m, to = m\}). \end{array}$$

Properties Assume that m is an embedding-projection pair, then we can prove

$$mapE\langle T\rangle\ m^{op} \cdot mapE\langle T\rangle\ m \quad = \quad id$$

by fixed point induction. We confine ourselves to the interesting cases.

- **Case** $T = Id$:

$$\begin{array}{ll} & mapE\langle Id\rangle\ m^{op} \cdot mapE\langle Id\rangle\ m \\ = & \quad \{ \text{ definition of } mapE \ \} \\ & from\ m^{op} \cdot from\ m \\ = & \quad \{ \ from\ m^{op} = to\ m \ \} \\ & to\ m \cdot from\ m \\ = & \quad \{ \ m \text{ is an embedding-projection pair } \} \\ & id. \end{array}$$

- **Case** $T = F \underline{\to} G$:

$$\begin{array}{ll} & mapE\langle F \underline{\to} G\rangle\ m^{op} \cdot mapE\langle F \underline{\to} G\rangle\ m \\ = & \quad \{ \text{ definition of } mapE \ \} \\ & (mapE\langle F\rangle\ (m^{op})^{op} \to mapE\langle G\rangle\ m^{op}) \cdot (mapE\langle F\rangle\ m^{op} \to mapE\langle G\rangle\ m) \\ = & \quad \{ \ (m^{op})^{op} = m \ \} \\ & (mapE\langle F\rangle\ m \to mapE\langle G\rangle\ m^{op}) \cdot (mapE\langle F\rangle\ m^{op} \to mapE\langle G\rangle\ m) \\ = & \quad \{ \ (\to) \text{ difunctor } \} \\ & (mapE\langle F\rangle\ m^{op} \cdot mapE\langle F\rangle\ m) \to (mapE\langle G\rangle\ m^{op} \cdot mapE\langle G\rangle\ m) \\ = & \quad \{ \text{ ex hypothesi } \} \\ & id \to id \\ = & \quad \{ \ (\to) \text{ difunctor } \} \\ & id. \end{array}$$

Using similar calculations we can furthermore show that

$$mapE\langle T\rangle\ m \cdot mapE\langle T\rangle\ m^{op} \quad \sqsubseteq \quad id.$$

Both laws imply that $ep\{from = mapE\langle T\rangle\ m, to = mapE\langle T\rangle\ m^{op}\}$ is again an embedding-projection pair. Furthermore, one can show that the mapping function $\lambda m \,.\, ep\{from = mapE\langle T\rangle\ m, to = mapE\langle T\rangle\ m^{op}\}$ is the functorial action of T in the category $\mathcal{C}po^{e}$ of complete partial orders and embedding-projection pairs.

MPC-style definition The analysis above suggests that we can turn *mapE* into a MPC-style definition if we make *mapE* itself return an embedding-projection pair, rather than just the *from* function.

$$\begin{array}{lcl}
MapE\langle \mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \mathfrak{T} \to \star \\
MapE\langle \star\rangle\ T_1\ T_2 & = & EP\ T_1\ T_2 \\
MapE\langle \mathfrak{T} \times \mathfrak{U}\rangle\ T_1\ T_2 & = & MapE\langle \mathfrak{T}\rangle\ (Outl\ T_1)\ (Outl\ T_2) \times MapE\langle \mathfrak{U}\rangle\ (Outr\ T_1)\ (Outr\ T_2) \\
MapE\langle \mathfrak{T} \to \mathfrak{U}\rangle\ T_1\ T_2 & = & \forall A_1\ A_2 \,.\, MapE\langle \mathfrak{T}\rangle\ A_1\ A_2 \to MapE\langle \mathfrak{U}\rangle\ (T_1\ A_1)\ (T_2\ A_2)
\end{array}$$

$$\begin{array}{lcl}
mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & MapE\langle \mathfrak{T}\rangle\ T\ T \\
mapE\langle\!\langle 1\rangle\!\rangle & = & idE \\
mapE\langle\!\langle Char\rangle\!\rangle & = & idE \\
mapE\langle\!\langle Int\rangle\!\rangle & = & idE \\
mapE\langle\!\langle +\rangle\!\rangle\ mA\ mB & = & ep\{from = from\ mA + from\ mB, to = to\ mA + to\ mB\} \\
mapE\langle\!\langle \times\rangle\!\rangle\ mA\ mB & = & ep\{from = from\ mA \times from\ mB, to = to\ mA \times to\ mB\} \\
mapE\langle\!\langle \to\rangle\!\rangle\ mA\ mB & = & ep\{from = to\ mA \to from\ mB, to = from\ mA \to to\ mB\}
\end{array}$$

Note that $mapE\langle F\rangle\ m = from\ (mapE\langle\!\langle F\rangle\!\rangle\ m)$.

Properties Let us briefly sketch the proof that $mapE\langle\!\langle F\rangle\!\rangle$ is indeed the functorial action of F in the category $\mathcal{C}po^{e}$. First, we have to show that $mapE\langle\!\langle F\rangle\!\rangle$ takes embedding-projection pairs to embedding-projection pairs. The logical relation $\mathcal{EP}$ generalizes this property to types of arbitrary kinds.

$$\begin{array}{lcl}
\mathcal{EP}\langle \mathfrak{T}\rangle\ T_1\ T_2 & \subseteq & MapE\langle \mathfrak{T}\rangle\ T_1\ T_2 \\
m \in \mathcal{EP}\langle \star\rangle\ T_1\ T_2 & \equiv & to\ m \cdot from\ m = id :: T_1 \to T_1\ \cap\ from\ m \cdot to\ m \sqsubseteq id :: T_2 \to T_2
\end{array}$$

We have $mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle \in \mathcal{EP}\langle \mathfrak{T}\rangle\ T\ T$. Second, we have to prove that $mapE\langle\!\langle F\rangle\!\rangle$ preserves identity. The generic version of this law states $mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle \in \mathcal{I}d\langle \mathfrak{T}\rangle\ T$ where $\mathcal{I}d$ is given by

$$\begin{array}{lcl}
\mathcal{I}d\langle \mathfrak{T}\rangle\ T & \subseteq & MapE\langle \mathfrak{T}\rangle\ T\ T \\
m \in \mathcal{I}d\langle \star\rangle\ T & \equiv & m = idE :: EP\ T\ T.
\end{array}$$

Third, it remains to prove that $mapE\langle\!\langle F\rangle\!\rangle$ respects composition. In general, we have $(mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle) \in \mathcal{C}omp\langle \mathfrak{T}\rangle\ T\ T\ T$ where

$$\begin{array}{l}
\mathcal{C}omp\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_3 \subseteq MapE\langle \mathfrak{T}\rangle\ T_2\ T_3 \times MapE\langle \mathfrak{T}\rangle\ T_1\ T_2 \times MapE\langle \mathfrak{T}\rangle\ T_1\ T_3 \\
(m_1, m_2, m_3) \in \mathcal{C}omp\langle \star\rangle\ T_1\ T_2\ T_3 \equiv m_1 \circ m_2 = m_3 :: EP\ T_1\ T_3.
\end{array}$$

Her is the proof of the last law (we confine ourselves to the interesting case):

- **Case** $T = (\to)$: We have to show that

$$\begin{array}{l}
(mapE\langle\!\langle \to\rangle\!\rangle, mapE\langle\!\langle \to\rangle\!\rangle, mapE\langle\!\langle \to\rangle\!\rangle) \in \mathcal{C}omp\langle \star \to \star \to \star\rangle\ (\to)\ (\to)\ (\to) \\
\quad \equiv\ mA_1 \circ mA_2 = mA_3\ \supset\ mB_1 \circ mB_2 = mB_3 \\
\qquad\quad \supset\ mapE\langle\!\langle \to\rangle\!\rangle\ mA_1\ mB_1 \circ mapE\langle\!\langle \to\rangle\!\rangle\ mA_2\ mB_2 = mapE\langle\!\langle \to\rangle\!\rangle\ mA_3\ mB_3
\end{array}$$

Note that $mA_1 \circ mA_2 = mA_3$ implies $to\ mA_2 \cdot to\ mA_1 = to\ mA_3$ (mA_1 and mA_2 are swapped) and $from\ mA_1 \cdot from\ mA_2 = from\ mA_3$. We reason

$$
\begin{array}{ll}
& mapE\langle\!\langle\rightarrow\rangle\!\rangle\ mA_1\ mB_1 \circ mapE\langle\!\langle\rightarrow\rangle\!\rangle\ mA_2\ mB_2 \\
= & \quad \{\ \text{definition of } mapE\ \} \\
& ep\{from = to\ mA_1 \rightarrow from\ mB_1, to = \underline{\quad}\} \circ ep\{from = to\ mA_2 \rightarrow from\ mB_2, to = \underline{\quad}\} \\
= & \quad \{\ \text{definition of } (\circ)\ \} \\
& ep\{from = (to\ mA_1 \rightarrow from\ mB_1) \cdot (to\ mA_2 \rightarrow from\ mB_2), to = \underline{\quad}\} \\
= & \quad \{\ (\rightarrow) \text{ difunctor}\ \} \\
& ep\{from = (to\ mA_2 \cdot to\ mA_1) \rightarrow (from\ mB_1 \cdot from\ mB_2), to = \underline{\quad}\} \\
= & \quad \{\ \text{assumptions: } to\ mA_2 \cdot to\ mA_1 = to\ mA_3 \text{ and } from\ mB_1 \cdot from\ mB_2 = from\ mB_3\ \} \\
& ep\{from = to\ mA_3 \rightarrow from\ mB_3, to = \underline{\quad}\} \\
= & \quad \{\ \text{definition of } mapE\ \} \\
& mapE\langle\!\langle\rightarrow\rangle\!\rangle\ mA_3\ mB_3.
\end{array}
$$

5.2.2 Monadic maps

Recall the definition of monads given in Section 2.2.2. Each monad gives rise to a category, the so-called *Kleisli category of a monad*, whose arrows are procedures. The identity arrow of the Kleisli category is given by *return* and composition is given by '$\diamond$'. To define the monadic mapping functions it is useful to lift $(+)$ and $(\times)$ to procedures. The pendant of $(+)$ is given by

$$
\begin{array}{lcl}
(\boxplus) & :: & \forall M\,.\,(Monad\ M) \Rightarrow \forall A_1\ A_2\,.\,(A_1 \rightarrow M\ A_2) \\
& & \qquad \rightarrow \forall B_1\ B_2\,.\,(B_1 \rightarrow M\ B_2) \\
& & \qquad\qquad \rightarrow ((A_1 + B_1) \rightarrow M\ (A_2 + B_2)) \\
(m \boxplus n)\ (inl\ a_1) & = & m\ a_1 \gg\!= \lambda a_2 \rightarrow return\ (inl\ a_2) \\
(m \boxplus n)\ (inr\ b_1) & = & n\ b_1 \gg\!= \lambda b_2 \rightarrow return\ (inr\ b_2)
\end{array}
$$

It easy to see that

$$
\begin{array}{lcl}
return \boxplus return & = & return \qquad (5.1) \\
(m_1 \diamond m_2) \boxplus (n_1 \diamond n_2) & = & (m_1 \boxplus n_1) \diamond (m_2 \boxplus n_2). \qquad (5.2)
\end{array}
$$

Thus, $(+)$ is a bifunctor over the Kleisli category. For products there is a choice to be made.

$$
\begin{array}{lcl}
(\boxslash), (\boxbslash) & :: & \forall M\,.\,(Monad\ M) \Rightarrow \forall A_1\ A_2\,.\,(A_1 \rightarrow M\ A_2) \\
& & \qquad \rightarrow \forall B_1\ B_2\,.\,(B_1 \rightarrow M\ B_2) \\
& & \qquad\qquad \rightarrow ((A_1 \times B_1) \rightarrow M\ (A_2 \times B_2)) \\
(m \boxslash n)\ (a_1, b_1) & = & m\ a_1 \gg\!= \lambda a_2 \rightarrow n\ b_1 \gg\!= \lambda b_2 \rightarrow return\ (a_2, b_2) \\
(m \boxbslash n)\ (a_1, b_1) & = & n\ b_1 \gg\!= \lambda b_2 \rightarrow m\ a_1 \gg\!= \lambda a_2 \rightarrow return\ (a_2, b_2)
\end{array}
$$

We can either execute $m\ a_1$ first and then $n\ b_1$ or vice versa. The symbols, '$\boxslash$' and '$\boxbslash$', have been chosen to indicate which component of the tuple is executed first. Again, it is straightforward to show that

$$
\begin{array}{lcl}
return \boxslash return & = & return \qquad (5.3) \\
return \boxbslash return & = & return. \qquad (5.4)
\end{array}
$$

However, both $(\boxslash)$ and $(\boxbslash)$ fail to preserve monadic composition, which implies that $(\times)$ is not a bifunctor.

POPL-style definition We have two monadic 'mapping functions', one which traverses the data structure from left to right, *mapMl*, and one which traverses the data structure from right to left, *mapMr*.

$$
\begin{array}{lcl}
mapMl\langle T :: \star \to \star\rangle & :: & \forall M\,.\,(Monad\ M) \Rightarrow \forall A_1\ A_2\,.\,(A_1 \to M\ A_2) \to (T\ A_1 \to M\ (T\ A_2)) \\
mapMl\langle Id\rangle\ m & = & m \\
mapMl\langle \underline{1}\rangle\ m & = & return \\
mapMl\langle \underline{Char}\rangle\ m & = & return \\
mapMl\langle \underline{Int}\rangle\ m & = & return \\
mapMl\langle F \mathbin{\underline{+}} G\rangle\ m & = & mapMl\langle F\rangle\ m \boxplus mapMl\langle G\rangle\ m \\
mapMl\langle F \mathbin{\underline{\times}} G\rangle\ m & = & mapMl\langle F\rangle\ m \boxslash mapMl\langle G\rangle\ m
\end{array}
$$

The definition of *mapMr* is identical to *mapMl* except for the '$\times$' case:

$$mapMr\langle F \mathbin{\underline{\times}} G\rangle\ m \quad = \quad mapMl\langle F\rangle\ m \boxbslash mapMl\langle G\rangle\ m.$$

A special case of *mapMl* is *threadl*, which threads a monad through a structure.

$$
\begin{array}{lcl}
threadl\langle F :: \star \to \star\rangle & :: & \forall M\,.\,(Monad\ M) \Rightarrow \forall A\,.\,F\ (M\ A) \to M\ (F\ A) \\
threadl\langle F\rangle & = & mapMl\langle F\rangle\ id
\end{array}
$$

An application Using the two monadic mapping functions we can, for instance, separate a container into its *shape* and its *contents*, see, (Jansson and Jeuring 2000). Briefly, a container of type $F\ A$ can be uniqely represented by its shape of type $F\ ()$ and its contents of type $[A]$. Separating into shape and contents may be useful, for instance, prior to data compression. Instead of compression a value of type, say, $F\ String$ directly, we first separate it and then compress the shape and the contents separately, the former possibly using structured methods (for instance, *encode*) and the latter using statistical methods.

To implement *separate* and *combine*, we use the state transformer monad defined in Section 2.2.2.

$$
\begin{array}{lcl}
separate\langle F :: \star \to \star\rangle & :: & \forall A\,.\,F\ A \to StateT\ [A]\ (F\ ()) \\
separate\langle F\rangle & = & mapMr\langle F\rangle\ put \\
combine\langle F :: \star \to \star\rangle & :: & \forall A\,.\,F\ () \to StateT\ [A]\ (F\ A) \\
combine\langle F\rangle & = & mapMl\langle F\rangle\ get
\end{array}
$$

The helper functions *put* and *get* are given by

$$
\begin{array}{lcl}
put & :: & \forall A\,.\,A \to StateT\ [A]\ () \\
put\ a & = & StateT\ (\lambda s \to ((), a : s)) \\
get & :: & \forall A\,.\,() \to StateT\ [A]\ A \\
get\ () & = & StateT\ (\lambda(a : s) \to (a, s)).
\end{array}
$$

Thus, *separate* traverses a given container of type $F\ A$, replaces every element of type A by $()$ and additionally adds the element as a side-effect to the list of elements that is maintained as the state. Its inverse, *combine*, puts the elements from the list into the slots of the container. Note that *separate* uses *mapMr* while *combine* uses *mapMl*. Since they use opposite traversals, we can prove that $separate\langle F\rangle \diamond combine\langle F\rangle = return$. We will show below that

$$mr \diamond ml = return \quad \supset \quad mapMr\langle F\rangle\ mr \diamond mapMl\langle F\rangle\ ml = return.$$

Consequently, it suffices to establish that $put \diamond get = return$.

$$
\begin{array}{ll}
 & (put \diamond get)\ a \\
= & \quad \{ \text{definition of } (\diamond) \} \\
 & put\ a \ggg\!= get \\
= & \quad \{ \text{definition of } (\ggg\!=) \} \\
 & StateT\ (\lambda s \to \mathbf{let}\ (x, s') = applyST\ (put\ a)\ s\ \mathbf{in}\ applyST\ (get\ x)\ s') \\
= & \quad \{ \text{definition of } applyST \text{ and } put \} \\
 & StateT\ (\lambda s \to applyST\ (get\ ())\ (a : s)) \\
= & \quad \{ \text{definition of } applyST \text{ and } get \} \\
 & StateT\ (\lambda s \to (a, s)) \\
= & \quad \{ \text{definition of } return \} \\
 & return\ a.
\end{array}
$$

MPC-style definition The generalization of *mapMl* and *mapMr* to types of arbitrary kinds is straightforward.

$$
\begin{array}{lcl}
MapM_M\langle \mathfrak{T} :: \square \rangle & :: & \mathfrak{T} \to \mathfrak{T} \to \star \\
MapM_M\langle \star \rangle\ T_1\ T_2 & = & T_1 \to M\ T_2 \\
MapM_M\langle \mathfrak{T} \times \mathfrak{U} \rangle\ T_1\ T_2 & = & MapM_M\langle \mathfrak{T} \rangle\ (Outl\ T_1)\ (Outl\ T_2) \\
 & & \quad \times\ MapM_M\langle \mathfrak{U} \rangle\ (Outr\ T_1)\ (Outr\ T_2) \\
MapM_M\langle \mathfrak{T} \to \mathfrak{U} \rangle\ T_1\ T_2 & = & \forall A_1\ A_2 \,.\, MapM_M\langle \mathfrak{T} \rangle\ A_1\ A_2 \\
 & & \qquad \to MapM_M\langle \mathfrak{U} \rangle\ (T_1\ A_1)\ (T_2\ A_2) \\
mapMl\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \forall M \,.\, (Monad\ M) \Rightarrow MapM_M\langle \mathfrak{T} \rangle\ T\ T \\
mapMl\langle\!\langle 1 \rangle\!\rangle & = & return \\
mapMl\langle\!\langle Char \rangle\!\rangle & = & return \\
mapMl\langle\!\langle Int \rangle\!\rangle & = & return \\
mapMl\langle\!\langle + \rangle\!\rangle\ mA\ mB & = & mA \boxplus mB \\
mapMl\langle\!\langle \times \rangle\!\rangle\ mA\ mB & = & mA \boxslash mB
\end{array}
$$

The type of the monadic mapping function makes use of a simple extension: *MapM* takes an additional type parameter, the underlying monad M, that is passed unchanged to the base case. One can safely think of M as a type parameter that is global to the definition.

A property Strictly speaking, *mapMl* and *mapMr* do not classify as mapping functions as they fail to preserve composition (recall that $(\times)$ is not a bifunctor). They satisfy, however, the following *inversion law*:

$$mr \diamond ml = return \quad \supset \quad mapMr\langle\!\langle F \rangle\!\rangle\ mr \diamond mapMl\langle\!\langle F \rangle\!\rangle\ ml = return.$$

The generalization of the inversion law to types of arbitrary kinds states that $(mapMr\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle, mapMl\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle) \in \mathcal{MI}nv_M\langle \mathfrak{T} \rangle\ T\ T$ where $\mathcal{MI}nv$ is given by

$$
\begin{array}{lcl}
\mathcal{MI}nv_M\langle \mathfrak{T} \rangle\ T_1\ T_2 & \subseteq & MapM_M\langle \mathfrak{T} \rangle\ T_1\ T_2 \times MapM_M\langle \mathfrak{T} \rangle\ T_2\ T_1 \\
(mr, ml) \in \mathcal{MI}nv_M\langle \star \rangle\ T_1\ T_2 & \equiv & mr \diamond ml = return :: T_1 \to M\ T_1.
\end{array}
$$

Note that the relation $\mathcal{MI}nv$ is pointed if *return* and $(\ggg\!=)$ are strict: we have $\bot \diamond \bot = return :: 0 \to M\ 0 \equiv \bot \ggg\!= \bot = return\ \bot :: M\ 0$.

- **Case** $T = C \in \{1, \mathit{Char}, \mathit{Int}\}$: We have to show that

$$\begin{aligned} &(\mathit{mapMr}\langle\!\langle C \rangle\!\rangle, \mathit{mapMl}\langle\!\langle C \rangle\!\rangle) \in \mathcal{MInv}_M\langle\star\rangle\ C\ C \\ &\quad\equiv\ \mathit{return} \diamond \mathit{return} = \mathit{return}, \end{aligned}$$

which holds.

- **Case** $T = (+)$: We have to show that

$$\begin{aligned} &(\mathit{mapMr}\langle\!\langle + \rangle\!\rangle, \mathit{mapMl}\langle\!\langle + \rangle\!\rangle) \in \mathcal{MInv}_M\langle\star \to \star \to \star\rangle\ (+)\ (+) \\ &\quad\equiv\ \mathit{mrA} \diamond \mathit{mlA} = \mathit{return} \supset \mathit{mrB} \diamond \mathit{mlB} = \mathit{return} \\ &\qquad\quad \supset (\mathit{mrA} \boxplus \mathit{mrB}) \diamond (\mathit{mlA} \boxplus \mathit{mlB}) = \mathit{return}. \end{aligned}$$

We reason as follows:

$$\begin{aligned} & (\mathit{mrA} \boxplus \mathit{mrB}) \diamond (\mathit{mlA} \boxplus \mathit{mlB}) \\ = & \quad \{ \text{ property (5.2): } (\boxplus) \text{ preserves } (\diamond) \ \} \\ & (\mathit{mrA} \diamond \mathit{mlA}) \boxplus (\mathit{mrB} \diamond \mathit{mlB}) \\ = & \quad \{ \text{ assumptions: } \mathit{mrA} \diamond \mathit{mlA} = \mathit{return} \text{ and } \mathit{mrB} \diamond \mathit{mlB} = \mathit{return} \ \} \\ & \mathit{return} \boxplus \mathit{return} \\ = & \quad \{ \text{ property (5.1): } (\boxplus) \text{ preserves } \mathit{return} \ \} \\ & \mathit{return}. \end{aligned}$$

- **Case** $T = (\times)$: We have to show that

$$\begin{aligned} &(\mathit{mapMr}\langle\!\langle \times \rangle\!\rangle, \mathit{mapMl}\langle\!\langle \times \rangle\!\rangle) \in \mathcal{MInv}_M\langle\star \to \star \to \star\rangle\ (\times)\ (\times) \\ &\quad\equiv\ \mathit{mrA} \diamond \mathit{mlA} = \mathit{return} \supset \mathit{mrB} \diamond \mathit{mlB} = \mathit{return} \\ &\qquad\quad \supset (\mathit{mrA} \boxbslash \mathit{mrB}) \diamond (\mathit{mlA} \boxslash \mathit{mlB}) = \mathit{return} \end{aligned}$$

This statement is most conveniently shown if we rephrase it using the so-called **do**-*notation*, see, for instance (Bird 1998). As an example, using the **do**-notation $\mathit{mr} \diamond \mathit{ml} = \mathit{return}$ reads

$$\mathbf{do}\ \{p; y \leftarrow \mathit{mr}\ x; z \leftarrow \mathit{ml}\ y; q; r\} = \mathbf{do}\ \{p; q[z := x]; r[z := x]\},$$

provided y is not free in q or r. Now, we reason:

$$\begin{aligned} & \mathbf{do}\ \{((\mathit{mrA} \boxbslash \mathit{mrB}) \diamond (\mathit{mlA} \boxslash \mathit{mlB}))\ (a_1, a_2)\} \\ = & \quad \{ \text{ definition of } (\diamond) \text{ and monad laws } \} \\ & \mathbf{do}\ \{b \leftarrow (\mathit{mrA} \boxbslash \mathit{mrB})\ (a_1, a_2); (\mathit{mlA} \boxslash \mathit{mlB})\ b\} \\ = & \quad \{ \text{ definition of } (\boxbslash) \text{ and monad laws } \} \\ & \mathbf{do}\ \{b_2 \leftarrow \mathit{mrB}\ a_2; b_1 \leftarrow \mathit{mrA}\ a_1; (\mathit{mlA} \boxslash \mathit{mlB})\ (b_1, b_2)\} \\ = & \quad \{ \text{ definition of } (\boxslash) \text{ and monad laws } \} \\ & \mathbf{do}\ \{b_2 \leftarrow \mathit{mrB}\ a_2; b_1 \leftarrow \mathit{mrA}\ a_1; c_1 \leftarrow \mathit{mlA}\ b_1; c_2 \leftarrow \mathit{mlB}\ b_2; \mathit{return}\ (c_1, c_2)\} \\ = & \quad \{ \text{ assumption: } \mathit{mrA} \diamond \mathit{mlA} = \mathit{return} \ \} \\ & \mathbf{do}\ \{b_2 \leftarrow \mathit{mrB}\ a_2; c_2 \leftarrow \mathit{mlB}\ b_2; \mathit{return}\ (a_1, c_2)\} \\ = & \quad \{ \text{ assumption: } \mathit{mrB} \diamond \mathit{mlB} = \mathit{return} \ \} \\ & \mathbf{do}\ \{\mathit{return}\ (a_1, a_2)\}. \end{aligned}$$

Remark 5.1 The development above can be generalized even further using the framework of *arrows* introduced by Hughes (2000). Though more abstract arrows simplify the calculations as demonstrated convincingly by Jansson and Jeuring (2000). □

5.3 Zipping functions

POPL-style definitions Closely related to mapping functions are zipping functions. A binary zipping function takes two structures of the same shape and combines them into a single structure. For instance, the list *zip* takes two lists of type $List\ A_1$ and $List\ A_2$ and pairs corresponding elements producing a list of type $List\ (A_1 \times A_2)$. The definition of *zip* is similar to that of *equal*.

$$
\begin{array}{lcl}
zip\langle T :: \star \to \star\rangle & :: & \forall A\ B\,.\,T\ A \times T\ B \to T\ (A \times B) \\
zip\langle Id\rangle\ (a, b) & = & (a, b) \\
zip\langle \underline{1}\rangle\ ((), ()) & = & () \\
zip\langle \underline{Char}\rangle\ (c_1, c_2) & = & zipChar\ (c_1, c_2) \\
zip\langle \underline{Int}\rangle\ (i_1, i_2) & = & zipInt\ (i_1, i_2) \\
zip\langle F \mathbin{\underline{+}} G\rangle\ (inl\ f_1, inl\ f_2) & = & inl\ (zip\langle F\rangle\ (f_1, f_2)) \\
zip\langle F \mathbin{\underline{+}} G\rangle\ (inl\ f_1, inr\ g_2) & = & error\ \texttt{"zip"} \\
zip\langle F \mathbin{\underline{+}} G\rangle\ (inr\ g_1, inl\ f_2) & = & error\ \texttt{"zip"} \\
zip\langle F \mathbin{\underline{+}} G\rangle\ (inr\ g_1, inr\ g_2) & = & inr\ (zip\langle G\rangle\ (g_1, g_2)) \\
zip\langle F \mathbin{\underline{\times}} G\rangle\ ((f_1, g_1), (f_2, g_2)) & = & (zip\langle F\rangle\ (f_1, f_2), zip\langle G\rangle\ (g_1, g_2))
\end{array}
$$

The helper functions *zipChar* and *zipInt* are defined

$$
\begin{array}{lcl}
zipChar & :: & Char \times Char \to Char \\
zipChar\ (c_1, c_2) & = & \mathbf{if}\ equalChar\ c_1\ c_2\ \mathbf{then}\ c_1\ \mathbf{else}\ error\ \texttt{"zipChar"} \\
zipInt & :: & Int \times Int \to Int \\
zipInt\ (i_1, i_2) & = & \mathbf{if}\ equalInt\ i_1\ i_2\ \mathbf{then}\ i_1\ \mathbf{else}\ error\ \texttt{"zipInt"}.
\end{array}
$$

Since *zip* has a polymorphic type, it satisfies a *naturality law*, in fact, a generic naturality law.

$$zip\langle F\rangle \cdot (map\langle F\rangle\ m_1 \times map\langle F\rangle\ m_2) \quad = \quad map\langle F\rangle\ (m_1 \times m_2) \cdot zip\langle F\rangle \tag{5.5}$$

A colleague of *zip* is *zipWith*, which enjoys the following specification:

$$zipWith\langle F\rangle\ z \quad = \quad map\langle F\rangle\ z \cdot zip\langle F\rangle. \tag{5.6}$$

The *zipWith* function captures the common idiom of composing a *map* with a *zip*. The derivation of *zipWith* is left as an exercise to the reader; we present only the final result:

$$
\begin{array}{lcl}
zipWith\langle T :: \star \to \star\rangle & :: & \forall A\ B\ C\,.\,(A \times B \to C) \to (T\ A \times T\ B \to T\ C) \\
zipWith\langle Id\rangle\ z\ (a, b) & = & z\ (a, b) \\
zipWith\langle \underline{1}\rangle\ z\ ((), ()) & = & () \\
zipWith\langle \underline{Char}\rangle\ z\ (c_1, c_2) & = & zipChar\ (c_1, c_2) \\
zipWith\langle \underline{Int}\rangle\ z\ (i_1, i_2) & = & zipInt\ (i_1, i_2) \\
zipWith\langle F \mathbin{\underline{+}} G\rangle\ z\ (inl\ f_1)\ (inl\ f_2) & = & inl\ (zipWith\langle F\rangle\ z\ (f_1, f_2)) \\
zipWith\langle F \mathbin{\underline{+}} G\rangle\ z\ (inl\ f_1)\ (inr\ g_2) & = & error\ \texttt{"zipWith"} \\
zipWith\langle F \mathbin{\underline{+}} G\rangle\ z\ (inr\ g_1)\ (inl\ f_2) & = & error\ \texttt{"zipWith"} \\
zipWith\langle F \mathbin{\underline{+}} G\rangle\ z\ (inr\ g_1)\ (inr\ g_2) & = & inr\ (zipWith\langle G\rangle\ z\ (g_1, g_2)) \\
zipWith\langle F \mathbin{\underline{\times}} G\rangle\ z\ (f_1, g_1)\ (f_2, g_2) & = & (zipWith\langle F\rangle\ z\ (f_1, f_2), zipWith\langle G\rangle\ z\ (g_1, g_2)).
\end{array}
$$

Note that we can define *zip* in terms of *zipWith*:

$$zip\langle F\rangle \quad = \quad zipWith\langle F\rangle\ id.$$

The *zipWith* function satisfies two general fusion laws: *map-zipWith*-fusion and *zipWith-map*-fusion.

$$\begin{aligned} map\langle F\rangle\ m \cdot zipWith\langle F\rangle\ z \quad &= \quad zipWith\langle F\rangle\ (m \cdot z) \\ zipWith\langle F\rangle\ z \cdot (map\langle F\rangle\ m_1 \times map\langle F\rangle\ m_2) \quad &= \quad zipWith\langle F\rangle\ (z \cdot (m_1 \times m_2)). \end{aligned}$$

The first law is a simple consequence of the specification (5.6). The second law is a consequence of the naturality law (5.5). Conversely, the *zipWith* laws imply the *zip* laws.

MPC-style definitions The *zipWith* function can be easily generalized to higher-order kinds. Its type is essentially a three parameter variant of *Map*.

$$\begin{aligned}
&ZipWith\langle \mathfrak{T} :: \Box\rangle && :: \quad \mathfrak{T} \to \mathfrak{T} \to \mathfrak{T} \to \star \\
&ZipWith\langle \star\rangle\ T_1\ T_2\ T_3 && = \quad T_1 \times T_2 \to T_3 \\
&ZipWith\langle \mathfrak{T} \times \mathfrak{U}\rangle\ T_1\ T_2\ T_3 && = \quad ZipWith\langle \mathfrak{T}\rangle\ (Outl\ T_1)\ (Outl\ T_2)\ (Outl\ T_3) \\
& && \qquad \times ZipWith\langle \mathfrak{U}\rangle\ (Outr\ T_1)\ (Outr\ T_2)\ (Outr\ T_3) \\
&ZipWith\langle \mathfrak{T} \to \mathfrak{U}\rangle\ T_1\ T_2\ T_3 && = \quad \forall A_1\ A_2\ A_3\ .\ ZipWith\langle \mathfrak{T}\rangle\ A_1\ A_2\ A_3 \\
& && \qquad\qquad \to ZipWith\langle \mathfrak{U}\rangle\ (T_1\ A_1)\ (T_2\ A_2)\ (T_3\ A_3) \\
&zipWith\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle && :: \quad ZipWith\langle \mathfrak{T}\rangle\ T\ T\ T \\
&zipWith\langle\!\langle 1\rangle\!\rangle\ ((),()) && = \quad () \\
&zipWith\langle\!\langle Char\rangle\!\rangle\ (c_1, c_2) && = \quad zipChar\ (c_1, c_2) \\
&zipWith\langle\!\langle Int\rangle\!\rangle\ (i_1, i_2) && = \quad zipInt\ (i_1, i_2) \\
&zipWith\langle\!\langle +\rangle\!\rangle\ zA\ zB\ (inl\ a_1, inl\ a_2) && = \quad inl\ (zA\ (a_1, a_2)) \\
&zipWith\langle\!\langle +\rangle\!\rangle\ zA\ zB\ (inl\ a_1, inr\ b_2) && = \quad error\ \texttt{"zipWith"} \\
&zipWith\langle\!\langle +\rangle\!\rangle\ zA\ zB\ (inr\ b_1, inl\ a_2) && = \quad error\ \texttt{"zipWith"} \\
&zipWith\langle\!\langle +\rangle\!\rangle\ zA\ zB\ (inr\ b_1, inr\ b_2) && = \quad inr\ (zB\ (b_1, b_2)) \\
&zipWith\langle\!\langle \times\rangle\!\rangle\ zA\ zB\ ((a_1, b_1), (a_2, b_2)) && = \quad (zA\ (a_1, a_2), zB\ (b_1, b_2))
\end{aligned}$$

Properties The generalized version of *zipWith* satisfies generalized versions of the two fusion laws. The first law states that

$$(map\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, zipWith\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, zipWith\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle) \in \mathcal{F}use_1\langle \mathfrak{T}\rangle\ T\ T\ T\ T$$

where $\mathcal{F}use_1$ is given by

$$\begin{aligned}
&\mathcal{F}use_1\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_3\ T_4 \quad \subseteq \quad Map\langle \mathfrak{T}\rangle\ T_3\ T_4 \times ZipWith\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_3 \times ZipWith\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_4 \\
&(m, z, z') \in \mathcal{F}use_1\langle \star\rangle\ T_1\ T_2\ T_3\ T_4 \\
&\qquad\qquad \equiv \quad m \cdot z = z' :: T_1 \times T_2 \to T_4.
\end{aligned}$$

Similarly, the second law formalizes that

$$(zipWith\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, map\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, map\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, zipWith\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle) \in \mathcal{F}use_2\langle \mathfrak{T}\rangle\ T\ T\ T\ T\ T$$

where $\mathcal{F}use_2$ is defined

$$\begin{aligned}
&\mathcal{F}use_2\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_3\ T_4\ T_5 \quad \subseteq \quad ZipWith\langle \mathfrak{T}\rangle\ T_3\ T_4\ T_5 \times Map\langle \mathfrak{T}\rangle\ T_1\ T_3 \\
&\qquad\qquad\qquad\qquad \times Map\langle \mathfrak{T}\rangle\ T_2\ T_4 \times ZipWith\langle \mathfrak{T}\rangle\ T_1\ T_2\ T_5 \\
&(z, m_1, m_2, z') \in \mathcal{F}use_2\langle \star\rangle\ T_1\ T_2\ T_3\ T_4\ T_5 \\
&\qquad\qquad \equiv \quad z \cdot (m_1 \times m_2) = z' :: T_1 \times T_2 \to T_5.
\end{aligned}$$

We only show the first fusion law, the proof of the second law is left as an exercise to the reader. For the calculations it is helpful to rephrase *zipWith* in a point-free style (we abbreviate *error* `"zipWith"` by $\perp$):

$$\begin{array}{lcl}
\mathit{zipWith}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{ZipWith}\langle \mathfrak{T} \rangle\ T\ T\ T \\
\mathit{zipWith}\langle\!\langle 1 \rangle\!\rangle & = & \mathit{unit} \\
\mathit{zipWith}\langle\!\langle \mathit{Char} \rangle\!\rangle & = & \mathit{zipChar} \\
\mathit{zipWith}\langle\!\langle \mathit{Int} \rangle\!\rangle & = & \mathit{zipInt} \\
\mathit{zipWith}\langle\!\langle + \rangle\!\rangle\ zA\ zB & = & (((\mathit{inl} \cdot zA) \mathbin{\triangledown} \perp) \mathbin{\triangledown} (\perp \mathbin{\triangledown} (\mathit{inr} \cdot zB))) \cdot \mathit{dist} \\
\mathit{zipWith}\langle\!\langle \times \rangle\!\rangle\ zA\ zB & = & (zA \times zB) \cdot \mathit{transpose}
\end{array}$$

The function *dist* combines *distr* and *distl* defined in Section 2.3.7.

$$\begin{array}{lcl}
\mathit{dist} & :: & \forall A\ B\ C\ D\,.\,(A + B) \times (C + D) \to ((A \times C) + (A \times D)) + ((B \times C) + (B \times D)) \\
\mathit{dist} & = & (\mathit{distr} + \mathit{distr}) \cdot \mathit{distl}
\end{array}$$

The function *transpose* transposes a 2×2 matrix.

$$\begin{array}{lcl}
\mathit{transpose} & :: & \forall A\ B\ C\ D\,.\,(A \times B) \times (C \times D) \to (A \times C) \times (B \times D) \\
\mathit{transpose}\ ((a, b), (c, d)) & = & ((a, c), (b, d))
\end{array}$$

Now, for the proof:

- **Case** $T = C \in \{1, \mathit{Int}, \mathit{Char}\}$: We have to show that

  $$\begin{array}{l}
  (\mathit{map}\langle\!\langle C \rangle\!\rangle, \mathit{zipWith}\langle\!\langle C \rangle\!\rangle, \mathit{zipWith}\langle\!\langle C \rangle\!\rangle) \in \mathcal{F}\mathit{use}_1\langle \star \rangle\ C\ C\ C\ C \\
  \quad \equiv\ \mathit{map}\langle\!\langle C \rangle\!\rangle \cdot \mathit{zipWith}\langle\!\langle C \rangle\!\rangle = \mathit{zipWith}\langle\!\langle C \rangle\!\rangle,
  \end{array}$$

  which holds since $\mathit{map}\langle\!\langle C \rangle\!\rangle = \mathit{id}$.

- **Case** $T = (+)$: We have to show that

  $$\begin{array}{l}
  (\mathit{map}\langle\!\langle + \rangle\!\rangle, \mathit{zipWith}\langle\!\langle + \rangle\!\rangle, \mathit{zipWith}\langle\!\langle + \rangle\!\rangle) \in \mathcal{F}\mathit{use}_1\langle \mathfrak{T} \rangle\ (+)\ (+)\ (+)\ (+) \\
  \quad \equiv\ mA \cdot zA = zA' \supset mB \cdot zB = zB' \\
  \qquad \supset\ \mathit{map}\langle\!\langle + \rangle\!\rangle\ mA\ mB \cdot \mathit{zipWith}\langle\!\langle + \rangle\!\rangle\ zA\ zB = \mathit{zipWith}\langle\!\langle + \rangle\!\rangle\ zA'\ zB'.
  \end{array}$$

  We reason:

  $$\begin{array}{cl}
  & \mathit{map}\langle\!\langle + \rangle\!\rangle\ mA\ mB \cdot \mathit{zipWith}\langle\!\langle + \rangle\!\rangle\ zA\ zB \\
  = & \quad \{\ \text{definition of } \mathit{map} \text{ and definition of } \mathit{zipWith}\ \} \\
  & (mA + mB) \cdot (((\mathit{inl} \cdot zA) \mathbin{\triangledown} \perp) \mathbin{\triangledown} (\perp \mathbin{\triangledown} (\mathit{inr} \cdot zB))) \cdot \mathit{dist} \\
  = & \quad \{\ \triangledown\text{-fusion law and } f \cdot \perp = \perp\ \} \\
  & ((((mA + mB) \cdot \mathit{inl} \cdot zA) \mathbin{\triangledown} \perp) \mathbin{\triangledown} (\perp \mathbin{\triangledown} ((mA + mB) \cdot \mathit{inr} \cdot zB))) \cdot \mathit{dist} \\
  = & \quad \{\ +\text{-computation law}\ \} \\
  & (((\mathit{inl} \cdot mA \cdot zA) \mathbin{\triangledown} \perp) \mathbin{\triangledown} (\perp \mathbin{\triangledown} (\mathit{inr} \cdot mB \cdot zB))) \cdot \mathit{dist} \\
  = & \quad \{\ \text{assumptions: } mA \cdot zA = zA' \text{ and } mB \cdot zB = zB'\ \} \\
  & (((\mathit{inl} \cdot zA') \mathbin{\triangledown} \perp) \mathbin{\triangledown} (\perp \mathbin{\triangledown} (\mathit{inr} \cdot zB'))) \cdot \mathit{dist} \\
  = & \quad \{\ \text{definition of } \mathit{zipWith}\ \} \\
  & \mathit{zipWith}\langle\!\langle + \rangle\!\rangle\ zA'\ zB'.
  \end{array}$$

- **Case** $T = (\times)$: We have to show that

$$\begin{aligned}
&(map\langle\!\langle \times \rangle\!\rangle, zipWith\langle\!\langle \times \rangle\!\rangle, zipWith\langle\!\langle \times \rangle\!\rangle) \in \mathcal{F}use_1\langle \mathfrak{T} \rangle\ (\times)\ (\times)\ (\times)\ (\times) \\
&\quad \equiv\ mA \cdot zA = zA' \supset mB \cdot zB = zB' \\
&\qquad\quad \supset\ map\langle\!\langle \times \rangle\!\rangle\ mA\ mB \cdot zipWith\langle\!\langle \times \rangle\!\rangle\ zA\ zB = zipWith\langle\!\langle \times \rangle\!\rangle\ zA'\ zB'.
\end{aligned}$$

We reason:

$$\begin{aligned}
& map\langle\!\langle \times \rangle\!\rangle\ mA\ mB \cdot zipWith\langle\!\langle \times \rangle\!\rangle\ zA\ zB \\
=\ & \quad \{\text{ definition of } map \text{ and definition of } zipWith\ \} \\
& (mA \times mB) \cdot (zA \times zB) \cdot transpose \\
=\ & \quad \{\ (\times) \text{ bifunctor }\} \\
& ((mA \cdot zA) \times (mB \cdot zB)) \cdot transpose \\
=\ & \quad \{\text{ assumptions: } mA \cdot zA = zA' \text{ and } mB \cdot zB = zB'\ \} \\
& (zA' \times zB') \cdot transpose \\
=\ & \quad \{\text{ definition of } zipWith\ \} \\
& zipWith\langle\!\langle \times \rangle\!\rangle\ zA'\ zB'.
\end{aligned}$$

Remark 5.2 The Haskell Prelude defines curried versions of *zip* and *zipWith*:

$$\begin{aligned}
zip &:: \forall A\ B\,.\,[A] \to [B] \to [A \times B] \\
zipWith &:: \forall A\ B\ C\,.\,(A \to B \to C) \to [A] \to [B] \to [C].
\end{aligned}$$

The curried versions are usually preferable for programming while the uncurried versions are preferable for conducting proofs. □

A variation The result of *zip* is a partial structure if the two arguments have not the same shape. Alternatively, one can define a zipping function of type $\forall A\ B\,.\,T\ A \to T\ B \to Maybe\ (T\ (A \times B))$, which uses the exception monad *Maybe* to signal incompatibility of the argument structures.

$$\begin{aligned}
& zip\langle T :: \star \to \star \rangle && :: \forall A\ B\,.\,T\ A \to T\ B \to Maybe\ (T\ (A \times B)) \\
& zip\langle Id \rangle\ a\ b && = return\ (a, b) \\
& zip\langle \underline{1} \rangle\ ()\ () && = return\ () \\
& zip\langle \underline{Char} \rangle\ c_1\ c_2 && = zipChar\ c_1\ c_2 \\
& zip\langle \underline{Int} \rangle\ i_1\ i_2 && = zipInt\ i_1\ i_2 \\
& zip\langle F \mathbin{\underline{+}} G \rangle\ (inl\ f_1)\ (inl\ f_2) && = mmap\ inl\ (zip\langle F \rangle\ f_1\ f_2) \\
& zip\langle F \mathbin{\underline{+}} G \rangle\ (inl\ f_1)\ (inr\ g_2) && = fail\ \texttt{"zip"} \\
& zip\langle F \mathbin{\underline{+}} G \rangle\ (inr\ g_1)\ (inl\ f_2) && = fail\ \texttt{"zip"} \\
& zip\langle F \mathbin{\underline{+}} G \rangle\ (inr\ g_1)\ (inr\ g_2) && = mmap\ inr\ (zip\langle G \rangle\ g_1\ g_2) \\
& zip\langle F \mathbin{\underline{\times}} G \rangle\ (f_1, g_1)\ (f_2, g_2) && = zip\langle F \rangle\ f_1\ f_2 \ggg\!= \lambda x_1 \to \\
& && \ zip\langle G \rangle\ g_1\ g_2 \ggg\!= \lambda x_2 \to \\
& && \ return\ (x_1, x_2)
\end{aligned}$$

Note that this version of *zip* is curried. The helper functions *zipChar* and *zipInt* are now given by

$$\begin{aligned}
zipChar &:: Char \to Char \to Maybe\ Char \\
zipChar\ c_1\ c_2 &= \mathbf{if}\ equalChar\ c_1\ c_2\ \mathbf{then}\ return\ c_1\ \mathbf{else}\ fail\ \texttt{"zipChar"} \\
zipInt &:: Int \to Int \to Maybe\ Int \\
zipInt\ i_1\ i_2 &= \mathbf{if}\ equalInt\ i_1\ i_2\ \mathbf{then}\ return\ i_1\ \mathbf{else}\ fail\ \texttt{"zipInt"}.
\end{aligned}$$

The MPC-style definition of *zip* is left as an exercise to the reader.

5.4 Reductions

A *reduction* or a *crush* (Meertens 1996) is a function that collapses a structure of values of type A into a single value of type A. This section explains how to define reductions generically.

5.4.1 POPL-style reductions

We have already encountered a special instance of a reduction: the *size* function. Here is another instance: the *flatten* function that flattens a structure into a list of elements.

$$
\begin{array}{lcl}
\mathit{flatten}\langle T :: \star \to \star\rangle & :: & \forall A\,.\,T\ A \to [A] \\
\mathit{flatten}\langle \mathit{Id}\rangle\ a & = & [a] \\
\mathit{flatten}\langle \underline{1}\rangle\ () & = & [\,] \\
\mathit{flatten}\langle \underline{\mathit{Char}}\rangle\ c & = & [\,] \\
\mathit{flatten}\langle \underline{\mathit{Int}}\rangle\ i & = & [\,] \\
\mathit{flatten}\langle F \mathbin{\underline{+}} G\rangle\ (\mathit{inl}\ f) & = & \mathit{flatten}\langle F\rangle\ f \\
\mathit{flatten}\langle F \mathbin{\underline{+}} G\rangle\ (\mathit{inr}\ g) & = & \mathit{flatten}\langle G\rangle\ g \\
\mathit{flatten}\langle F \mathbin{\underline{\times}} G\rangle\ (f, g) & = & \mathit{flatten}\langle F\rangle\ f \mathbin{+\!\!+} \mathit{flatten}\langle G\rangle\ g
\end{array}
$$

Note that $\mathit{flatten}\langle T\rangle\ t$ yields the contents of the container t, see also Section 5.2.2.

The definitions of *size* and *flatten* exhibit a common pattern: the elements of a base type are replaced by a constant (0 and $[\,]$, respectively) and the pair constructor is replaced by a binary operator ($(+)$ and $(+\!\!+)$, respectively). The generic function *reduce* abstracts away from these particularities.

$$
\begin{array}{lcl}
\mathit{reduce}\langle T :: \star \to \star\rangle & :: & \forall A\,.\,A \to (A \to A \to A) \to (T\ A \to A) \\
\mathit{reduce}\langle \mathit{Id}\rangle\ e\ \mathit{op}\ a & = & a \\
\mathit{reduce}\langle \underline{1}\rangle\ e\ \mathit{op}\ () & = & e \\
\mathit{reduce}\langle \underline{\mathit{Char}}\rangle\ e\ \mathit{op}\ c & = & e \\
\mathit{reduce}\langle \underline{\mathit{Int}}\rangle\ e\ \mathit{op}\ i & = & e \\
\mathit{reduce}\langle F \mathbin{\underline{+}} G\rangle\ e\ \mathit{op}\ (\mathit{inl}\ f) & = & \mathit{reduce}\langle F\rangle\ e\ \mathit{op}\ f \\
\mathit{reduce}\langle F \mathbin{\underline{+}} G\rangle\ e\ \mathit{op}\ (\mathit{inr}\ g) & = & \mathit{reduce}\langle G\rangle\ e\ \mathit{op}\ g \\
\mathit{reduce}\langle F \mathbin{\underline{\times}} G\rangle\ e\ \mathit{op}\ (f, g) & = & \mathit{reduce}\langle F\rangle\ e\ \mathit{op}\ f\ \text{`}\mathit{op}\text{`}\ \mathit{reduce}\langle G\rangle\ e\ \mathit{op}\ g
\end{array}
$$

We can define *reduce* more succinctly using a local definition[1] and employing a point-free style.

$$
\begin{array}{lcl}
\mathit{reduce}\langle T :: \star \to \star\rangle & :: & \forall A\,.\,A \to (A \to A \to A) \to (T\ A \to A) \\
\mathit{reduce}\langle T\rangle\ e\ \mathit{op} & = & \mathit{red}\langle T\rangle \\
\quad \textbf{where}\ \mathit{red}\langle T :: \star \to \star\rangle & :: & T\ A \to A \\
\qquad\quad \mathit{red}\langle \mathit{Id}\rangle & = & \mathit{id} \\
\qquad\quad \mathit{red}\langle K\ C\rangle & = & k\ e \\
\qquad\quad \mathit{red}\langle F \mathbin{\underline{+}} G\rangle & = & \mathit{red}\langle F\rangle \mathbin{\triangledown} \mathit{red}\langle G\rangle \\
\qquad\quad \mathit{red}\langle F \mathbin{\underline{\times}} G\rangle & = & \mathit{uncurry}\ \mathit{op} \cdot (\mathit{red}\langle F\rangle \times \mathit{red}\langle G\rangle)
\end{array}
$$

A number of useful functions can be implemented in terms of *reduce* and *map*, see Figure 5.1. Meertens (1996), Jansson and Jeuring (1998) give further applications.

[1] We assume that type variables appearing in type signatures are scoped, that is, the type variable A in the signature of $\mathit{red}\langle T\rangle$ is *not* universally quantified but refers to the occurrence in *reduce*'s signature.

$$
\begin{array}{lcl}
sum\langle F :: \star \to \star\rangle & :: & \forall N \,.\, (Num\ N) \Rightarrow F\ N \to N \\
sum\langle F\rangle & = & reduce\langle F\rangle\ 0\ (+) \\
and\langle F :: \star \to \star\rangle & :: & F\ Bool \to Bool \\
and\langle F\rangle & = & reduce\langle F\rangle\ true\ (\wedge) \\
minimum\langle F :: \star \to \star\rangle & :: & \forall A \,.\, (Bounded\ A, Ord\ A) \Rightarrow F\ A \to A \\
minimum\langle F\rangle & = & reduce\langle F\rangle\ maxBound\ min \\
size\langle F :: \star \to \star\rangle & :: & \forall A \,.\, (Num\ N) \Rightarrow F\ A \to N \\
size\langle F\rangle & = & sum\langle F\rangle \cdot map\langle F\rangle\ (k\ 1) \\
all\langle F :: \star \to \star\rangle & :: & \forall A \,.\, (A \to Bool) \to (F\ A \to Bool) \\
all\langle F\rangle\ p & = & and\langle F\rangle \cdot map\langle F\rangle\ p \\
flatten\langle F :: \star \to \star\rangle & :: & \forall A \,.\, F\ A \to [A] \\
flatten\langle F\rangle & = & reduce\langle F\rangle\ [\,]\ (+\!\!+) \cdot map\langle F\rangle\ wrap \\
\mathbf{data}\ Shape\ A & = & Empty \mid Var\ A \mid Bin\ (Shape\ A)\ (Shape\ A) \\
shape\langle F :: \star \to \star\rangle & :: & \forall A \,.\, F\ A \to Shape\ A \\
shape\langle F\rangle & = & reduce\langle F\rangle\ Empty\ Bin \cdot map\langle F\rangle\ Var
\end{array}
$$

Figure 5.1: Examples of reductions (POPL-style).

5.4.2 MPC-style reductions

The MPC-style variant of *flatten* is similar to that of *size*.

$$
\begin{array}{lcl}
Flatten_Z\langle \mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \star \\
Flatten_Z\langle \star\rangle\ T & = & T \to [Z] \\
Flatten_Z\langle \mathfrak{T} \times \mathfrak{U}\rangle\ T & = & Flatten_Z\langle \mathfrak{T}\rangle\ (Outl\ T) \times Flatten_Z\langle \mathfrak{U}\rangle\ (Outr\ T) \\
Flatten_Z\langle \mathfrak{T} \to \mathfrak{U}\rangle\ T & = & \forall A \,.\, Flatten_Z\langle \mathfrak{T}\rangle\ A \to Flatten_Z\langle \mathfrak{U}\rangle\ (T\ A) \\
flatten\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & \forall Z \,.\, Flatten_Z\langle \mathfrak{T}\rangle\ T \\
flatten\langle\!\langle 1\rangle\!\rangle\ () & = & [\,] \\
flatten\langle\!\langle Char\rangle\!\rangle\ c & = & [\,] \\
flatten\langle\!\langle Int\rangle\!\rangle\ i & = & [\,] \\
flatten\langle\!\langle +\rangle\!\rangle\ flA\ flB\ (inl\ a) & = & flA\ a \\
flatten\langle\!\langle +\rangle\!\rangle\ flA\ flB\ (inr\ b) & = & flB\ b \\
flatten\langle\!\langle \times\rangle\!\rangle\ flA\ flB\ (a, b) & = & flA\ a +\!\!+ flB\ b
\end{array}
$$

Note that *flatten* is pointless for types—$flatten\langle\!\langle T\rangle\!\rangle\ t$ returns $[\,]$ for all types T of kind $\star$ provided t is finite and fully defined— but useful for type constructors. In particular, we can define the POPL-style *flatten* in terms of the MPC-style *flatten* (recall that $wrap\ a = [a]$):

$$
\begin{array}{lcl}
flatten\langle F :: \star \to \star\rangle & :: & \forall A \,.\, F\ A \to [A] \\
flatten\langle F\rangle & = & flatten\langle\!\langle F\rangle\!\rangle\ wrap.
\end{array}
$$

$$
\begin{array}{lcl}
sum\langle F :: \star \to \star\rangle & :: & \forall N\,.\,(Num\ N) \Rightarrow F\ N \to N \\
sum\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ 0\ (+)\ id \\
and\langle F :: \star \to \star\rangle & :: & F\ Bool \to Bool \\
and\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ true\ (\wedge)\ id \\
minimum\langle F :: \star \to \star\rangle & :: & \forall A\,.\,(Bounded\ A, Ord\ A) \Rightarrow F\ A \to A \\
minimum\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ maxBound\ min\ id \\
size\langle F :: \star \to \star\rangle & :: & \forall A\,.\,(Num\ N) \Rightarrow F\ A \to N \\
size\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ 0\ (+)\ (k\ 1) \\
all\langle F :: \star \to \star\rangle & :: & \forall A\,.\,(A \to Bool) \to (F\ A \to Bool) \\
all\langle F\rangle\ p & = & reduce\langle\!\langle F\rangle\!\rangle\ true\ (\wedge)\ p \\
flatten\langle F :: \star \to \star\rangle & :: & \forall A\,.\,F\ A \to [A] \\
flatten\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ [\,]\ (+\!\!+)\ wrap \\
biflatten\langle G :: \star \to \star \to \star\rangle & :: & \forall A\ B\,.\,G\ A\ B \to [A + B] \\
biflatten\langle G\rangle & = & reduce\langle\!\langle G\rangle\!\rangle\ [\,]\ (+\!\!+)\ (wrap \cdot inl)\ (wrap \cdot inr) \\
\mathbf{data}\ Shape\ A & = & Empty \mid Var\ A \mid Bin\ (Shape\ A)\ (Shape\ A) \\
shape\langle F :: \star \to \star\rangle & :: & \forall A\,.\,F\ A \to Tree\ A \\
shape\langle F\rangle & = & reduce\langle\!\langle F\rangle\!\rangle\ Empty\ Bin\ Var
\end{array}
$$

Figure 5.2: Examples of reductions (MPC-style).

Here is the generalized version of *reduce*.

$$
\begin{array}{lcl}
Reduce_Z\langle\mathfrak{T} :: \Box\rangle & :: & \mathfrak{T} \to \star \\
Reduce_Z\langle\star\rangle\ T & = & T \to Z \\
Reduce_Z\langle\mathfrak{T} \times \mathfrak{U}\rangle\ T & = & Reduce_Z\langle\mathfrak{T}\rangle\ (Outl\ T) \times Reduce_Z\langle\mathfrak{U}\rangle\ (Outr\ T) \\
Reduce_Z\langle\mathfrak{T} \to \mathfrak{U}\rangle\ T & = & \forall A\,.\,Reduce_Z\langle\mathfrak{T}\rangle\ A \to Reduce_Z\langle\mathfrak{U}\rangle\ (T\ A) \\
reduce\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & \forall Z\,.\,Z \to (Z \to Z \to Z) \to Reduce_Z\langle\mathfrak{T}\rangle\ T \\
reduce\langle\!\langle T\rangle\!\rangle\ e\ op & = & red\langle\!\langle T\rangle\!\rangle \\
\quad \mathbf{where} & & \\
\quad red\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & Reduce_Z\langle\mathfrak{T}\rangle\ T \\
\quad red\langle\!\langle 1\rangle\!\rangle & = & k\ e \\
\quad red\langle\!\langle Char\rangle\!\rangle & = & k\ e \\
\quad red\langle\!\langle Int\rangle\!\rangle & = & k\ e \\
\quad red\langle\!\langle +\rangle\!\rangle\ redA\ redb & = & redA\ \triangledown\ redb \\
\quad red\langle\!\langle \times\rangle\!\rangle\ redA\ redb & = & uncurry\ op \cdot (redA \times redb)
\end{array}
$$

The type of $reduce\langle\!\langle F\rangle\!\rangle$ where F is a unary type constructor is quite general.

$$reduce\langle\!\langle F :: \star \to \star\rangle\!\rangle \quad :: \quad \forall Z\,.\,Z \to (Z \to Z \to Z) \to (\forall A\,.\,(A \to Z) \to (F\ A \to Z))$$

Again, we can define the POPL-style *reduce* in terms of the MPC-style *reduce*.

$$
\begin{array}{lcl}
reduce\langle F :: \star \to \star\rangle & :: & \forall A\,.\,A \to (A \to A \to A) \to (F\ A \to A) \\
reduce\langle F\rangle\ e\ op & = & reduce\langle\!\langle F\rangle\!\rangle\ e\ op\ id
\end{array}
$$

Figure 5.2 lists some typical applications of $reduce\langle\!\langle F\rangle\!\rangle$ and $reduce\langle\!\langle G\rangle\!\rangle$ where G is a binary type constructor. Most of the definitions are obtained from Figure 5.1 using $reduce\langle F\rangle\ e\ op \cdot map\langle F\rangle\ m = reduce\langle\!\langle F\rangle\!\rangle\ e\ op\ m$. A generalization of this property will be proved in the following section.

5.4.3 Properties

Reductions satisfy two general fusion laws. The first law shows how to fuse a reduction with a map. The second law states conditions under which we can fuse an 'ordinary' function with a reduction.

The first law uses a logical relation that is a minor variant of *Comp*.

$$\begin{array}{lcl} \mathit{Comp}_Z\langle\mathfrak{T}\rangle\ T_1\ T_2 & \subseteq & \mathit{Reduce}_Z\langle\mathfrak{T}\rangle\ T_2 \times \mathit{Map}\langle\mathfrak{T}\rangle\ T_1\ T_2 \times \mathit{Reduce}_Z\langle\mathfrak{T}\rangle\ T_1 \\ (r, m, r') \in \mathit{Comp}_Z\langle\star\rangle\ T_1\ T_2 & \equiv & r \cdot m = r' :: T_1 \to Z \end{array}$$

Given an element $e :: Z$ and an operation $op :: Z \to Z \to Z$, we have

$$(\mathit{reduce}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\ e\ op, \mathit{map}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle, \mathit{reduce}\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\ e\ op) \in \mathit{Comp}_Z\langle\mathfrak{T}\rangle\ T\ T.$$

An immediate consequence of this property is (here $T :: \star \to \star$ is a unary type constructor)

$$\mathit{reduce}\langle\!\langle T\rangle\!\rangle\ e\ op\ f \cdot \mathit{map}\langle\!\langle T\rangle\!\rangle\ g = \mathit{reduce}\langle\!\langle T\rangle\!\rangle\ e\ op\ (f \cdot g),$$

which shows how to fuse a reduction with a map. As usual, to prove the generic property we merely have to verify that the statement holds for every type constant $C \in \mathit{Const}$. Using the point-free definitions of *map* and *red* this amounts to showing that

$$\begin{array}{lcl} k\ e \cdot id & = & k\ e \\ (r_1 \triangledown r_2) \cdot (m_1 + m_2) & = & (r_1 \cdot m_1) \triangledown (r_2 \cdot m_2) \\ \mathit{uncurry}\ op \cdot (r_1 \times r_2) \cdot (m_1 \times m_2) & = & \mathit{uncurry}\ op \cdot ((r_1 \cdot m_1) \times (r_2 \cdot m_2)). \end{array}$$

All three conditions hold.

Previous approaches to generic programming (Jansson and Jeuring 1997) required the programmer to specify the action of a generic function for the composition of two type constructors: for instance, for *size* the generic programmer had to supply the equation $\mathit{size}\langle F_1 \cdot F_2\rangle = \mathit{sum}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{size}\langle F_2\rangle)$. Interestingly, using *reduce-map* fusion this equation can be derived from the definitions of *size* and *sum* given in Figure 5.2.

$$\begin{array}{ll} & \mathit{size}\langle F_1 \cdot F_2\rangle \\ = & \quad \{ \text{definition of } \mathit{size} \} \\ & \mathit{reduce}\langle\!\langle F_1 \cdot F_2\rangle\!\rangle\ 0\ (+)\ (k\ 1) \\ = & \quad \{ \mathit{poly}\langle\!\langle F_1 \cdot F_2\rangle\!\rangle = \mathit{poly}\langle\!\langle F_1\rangle\!\rangle \cdot \mathit{poly}\langle\!\langle F_2\rangle\!\rangle \} \\ & \mathit{reduce}\langle\!\langle F_1\rangle\!\rangle\ 0\ (+)\ (\mathit{reduce}\langle\!\langle F_2\rangle\!\rangle\ 0\ (+)\ (k\ 1)) \\ = & \quad \{ \text{definition of } \mathit{size} \} \\ & \mathit{reduce}\langle\!\langle F_1\rangle\!\rangle\ 0\ (+)\ (\mathit{size}\langle F_2\rangle) \\ = & \quad \{ \mathit{reduce\text{-}map} \text{ fusion} \} \\ & \mathit{reduce}\langle\!\langle F_1\rangle\!\rangle\ 0\ (+)\ id \cdot \mathit{map}\langle F_1\rangle\ (\mathit{size}\langle F_2\rangle) \\ = & \quad \{ \text{definition of } \mathit{sum} \} \\ & \mathit{sum}\langle F_1\rangle \cdot \mathit{map}\langle F_1\rangle\ (\mathit{size}\langle F_2\rangle) \end{array}$$

The second law generalizes the fusion law for reductions given by Meertens (1996) to higher-order kinds. We have already derived a special instance of this law in Section 4.3.2. For that reason, we confine ourselves to a few remarks. The law is based on the logical relation $\mathcal{F}use$ defined by

$$\begin{array}{lcl}
\mathcal{F}use_{h,Z_1,Z_2}\langle\mathfrak{T}\rangle\ T & \subseteq & Reduce_{Z_1}\langle\mathfrak{T}\rangle\ T \times Reduce_{Z_2}\langle\mathfrak{T}\rangle\ T \\
(r, r') \in \mathcal{F}use_{h,Z_1,Z_2}\langle\star\rangle\ T & \equiv & h \cdot r = r' :: T \to Z_2,
\end{array}$$

where Z_1 and Z_2 are fixed types and $h :: Z_1 \to Z_2$ is a fixed function. The second fusion law, which gives conditions for fusing the function h with a reduction, then takes the following form:

$$\begin{array}{ll}
 & h\ \bot = \bot \\
\cap & h\ e = e' \\
\cap & h\ (op\ x\ y) = op'\ (h\ x)\ (h\ y) \\
\supset & (reduce\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\ e\ op, reduce\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle\ e'\ op') \in \mathcal{F}use_{h,Z_1,Z_2}\langle\mathfrak{T}\rangle\ T.
\end{array}$$

We can apply this law, for instance, to prove that $length \cdot flatten\langle F\rangle = size\langle F\rangle$, that is, $length \cdot reduce\langle\!\langle F\rangle\!\rangle\ [\,]\ (+\!\!+)\ wrap = reduce\langle\!\langle F\rangle\!\rangle\ 0\ (+)\ (k\ 1)$.

5.4.4 Right and left reductions

The implementations of *flatten* given in the previous sections have a quadratic running time since the computation of $x +\!\!+ y$ takes time proportional to the length of x. Using the well-known technique of *accumulation* (Bird 1998) we can improve the running time to $O(n)$. We have already used accumulation in Section 4.2 to derive an efficient implementation of *encode*. The following derivation is slightly more general in that it works for arbitrary operations under the proviso that op is associative and e is the unit of op.

The basic idea is to define a function $redr\langle F\rangle$ such that

$$op\ (red\langle F\rangle\ x)\ a \quad = \quad redr\langle F\rangle\ x\ a.$$

In a point-free style this condition can be written more succinctly as

$$op \cdot red\langle F\rangle \quad = \quad redr\langle F\rangle.$$

Now, assuming that *redr* itself can be expressed as a reduction we invoke the second fusion law for reductions:

$$op \cdot reduce\langle\!\langle F\rangle\!\rangle\ e\ op\ id \quad = \quad reduce\langle\!\langle F\rangle\!\rangle\ e'\ op'\ (op \cdot id).$$

Let us try to determine e' and op'. The fusion law requires them to satisfy $op\ e = e'$ and $op\ (op\ a\ b) = op'\ (op\ a)\ (op\ b)$. To derive e' we reason:

$$\begin{array}{ll}
 & op\ e \\
= & \quad \{\ \eta\text{-conversion}\ \} \\
 & \lambda x\,.\,op\ e\ x \\
= & \quad \{\ e \text{ is the unit of } op\text{: } op\ e\ x = x\ \} \\
 & \lambda x\,.\,x \\
= & \quad \{\ \text{definition of } id\ \} \\
 & id.
\end{array}$$

For op' we calculate:

$$\begin{array}{cl}
& op\ (op\ a\ b) \\
= & \quad \{\ \eta\text{-conversion}\ \} \\
& \lambda x\,.\,op\ (op\ a\ b)\ x \\
= & \quad \{\ op \text{ is associative: } op\ (op\ x\ y)\ z = op\ x\ (op\ y\ z)\ \} \\
& \lambda x\,.\,op\ a\ (op\ b\ x) \\
= & \quad \{\ \text{definition of } (\cdot)\ \} \\
& op\ a \cdot op\ b.
\end{array}$$

We have $e' = id$ and $op' = (\cdot)$. Consequently, we can define a more efficient POPL-style variant of *reduce* as follows:

$$\begin{array}{cl}
& reduce\langle F\rangle\ e\ op\ x \\
= & \quad \{\ e \text{ is the neutral element of } op\text{: } op\ x\ e = x\ \} \\
& op\ (reduce\langle F\rangle\ e\ op\ x)\ e \\
= & \quad \{\ \text{definition of } reduce\ \} \\
& op\ (reduce\langle\!\langle F\rangle\!\rangle\ e\ op\ id\ x)\ e \\
= & \quad \{\ \text{fusion, see above}\ \} \\
& reduce\langle\!\langle F\rangle\!\rangle\ id\ (\cdot)\ op\ x\ e
\end{array}$$

To summarize, we have derived the following definition:

$$\begin{array}{lcl}
reduce'\langle F :: \star \to \star\rangle & :: & \forall A\,.\,A \to (A \to A \to A) \to (F\ A \to A) \\
reduce'\langle F\rangle\ e\ op\ x & = & reduce\langle\!\langle F\rangle\!\rangle\ id\ (\cdot)\ op\ x\ e.
\end{array}$$

The implementation guarantees that applications of op are only nested to the right. For instance, if x contains from left to right the elements $a_1, \ldots, a_n$, then $reduce'\langle F\rangle\ e\ op\ x$ evaluates to

$$(op\ a_1 \cdot op\ a_2 \cdot \cdots \cdot op\ a_n)\ e \quad = \quad op\ a_1\ (op\ a_2\ (\ldots\ (op\ a_n\ e)\ \ldots)).$$

This property also reveals that the type of *reduce'* is unnecessarily restricted: the two arguments of op need not have the same type. Therefore, we may generalize the type signature as follows.

$$\begin{array}{lcl}
reducer\langle F :: \star \to \star\rangle & :: & \forall A\ B\,.\,(A \to B \to B) \to (F\ A \to B \to B) \\
reducer\langle F\rangle\ op & = & reduce\langle\!\langle F\rangle\!\rangle\ id\ (\cdot)\ op
\end{array}$$

Note that we have also rearranged the arguments to emphasize the structure.

Building upon $reducer\langle F\rangle$ we can now give a linear-time program for $flatten\langle F\rangle$.

$$\begin{array}{lcl}
flatten\langle F :: \star \to \star\rangle & :: & \forall A\,.\,F\ A \to [A] \\
flatten\langle F\rangle\ f & = & reducer\langle F\rangle\ (:)\ f\ [\,]
\end{array}$$

Of course, there is also a reduction to the left. We merely have to flip composition and the binary operation op.

$$\begin{array}{lcl}
reducel\langle F :: \star \to \star\rangle & :: & \forall A\ B\,.\,(B \to A \to B) \to (B \to F\ A \to B) \\
reducel\langle F\rangle\ op & = & flip\ (reduce\langle\!\langle F\rangle\!\rangle\ id\ (flip\ (\cdot))\ (flip\ op))
\end{array}$$

Writing *flip* $(\cdot)$ as $(;)$ we have

$$(\mathit{flip}\ op\ a_1 ; \mathit{flip}\ op\ a_2 ; \ldots ; \mathit{flip}\ op\ a_n)\ e \quad = \quad op\ (\ldots\ (op\ (op\ e\ a_1)\ a_2)\ \ldots)\ a_n .$$

The workings of *reducer* and *reducel* become more apparent if we partially evaluate the two definitions obtaining the following POPL-style implementations:

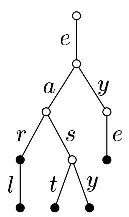
$$\begin{array}{lcl}
\mathit{reducer}\langle F :: \star \to \star \rangle & :: & \forall A\ B \,.\, (A \to B \to B) \to (F\ A \to B \to B) \\
\mathit{reducer}\langle \mathit{Id} \rangle\ op\ a\ b & = & op\ a\ b \\
\mathit{reducer}\langle K\ C \rangle\ op\ c\ b & = & b \\
\mathit{reducer}\langle F \underline{+} G \rangle\ op\ (\mathit{inl}\ f)\ b & = & \mathit{reducer}\langle F \rangle\ op\ f\ b \\
\mathit{reducer}\langle F \underline{+} G \rangle\ op\ (\mathit{inr}\ g)\ b & = & \mathit{reducer}\langle G \rangle\ op\ g\ b \\
\mathit{reducer}\langle F \underline{\times} G \rangle\ op\ (f, g)\ b & = & \mathit{reducer}\langle F \rangle\ op\ f\ (\mathit{reducer}\langle G \rangle\ op\ g\ b) \\
\mathit{reducel}\langle F :: \star \to \star \rangle & :: & \forall A\ B \,.\, (B \to A \to B) \to (B \to F\ A \to B) \\
\mathit{reducel}\langle \mathit{Id} \rangle\ op\ b\ a & = & op\ b\ a \\
\mathit{reducel}\langle K\ C \rangle\ op\ b\ c & = & b \\
\mathit{reducel}\langle F \underline{+} G \rangle\ op\ b\ (\mathit{inl}\ f) & = & \mathit{reducel}\langle F \rangle\ op\ b\ f \\
\mathit{reducel}\langle F \underline{+} G \rangle\ op\ b\ (\mathit{inr}\ g) & = & \mathit{reducel}\langle G \rangle\ op\ b\ g \\
\mathit{reducel}\langle F \underline{\times} G \rangle\ op\ b\ (f, g) & = & \mathit{reducel}\langle F \rangle\ op\ (\mathit{reducel}\langle G \rangle\ op\ b\ f)\ g .
\end{array}$$

5.5 Generic dictionaries

A trie is a search tree scheme that employs the structure of search keys to organize information. Tries were originally devised as a means to represent a collection of records indexed by strings over a fixed alphabet. Based on work by Wadsworth and others, Connelly and Morris (1995) generalized the concept to permit indexing by elements built according to an arbitrary signature. In this section we go one step further and define tries and operations on tries generically for arbitrary data types of arbitrary kinds, including parameterized and nested data types.

5.5.1 Introduction

The concept of a trie was introduced by Thue in 1912 as a means to represent a set of strings, see Knuth (1998). In its simplest form a trie is a multiway branching tree where each edge is labelled with a character. For example, the set of strings $\{\mathit{ear}, \mathit{earl}, \mathit{east}, \mathit{easy}, \mathit{eye}\}$ is represented by the trie depicted on the right. Searching in a trie starts at the root and proceeds by traversing the edge that matches the first character, then traversing the edge that matches the second character, and so forth. The search key is a member of the represented set if the search stops in a node that is marked—marked nodes are drawn as filled circles on the right. Tries can also be used to represent finite maps. In this case marked nodes additionally contain values associated with the strings. Interestingly, the move from sets to finite maps is not a mere variation of the scheme. As we shall see it is essential for the further development.



On a more abstract level a trie itself can be seen as a composition of finite maps. Each collection of edges descending from the same node constitutes a finite map sending a character to a trie. With this interpretation in mind it is relatively straightforward to devise an implementation of string-indexed tries. If strings are defined by the data type introduced in Section 1.1.1 (page 3)

$$\textbf{data}\ \mathit{String} \quad = \quad \mathit{nilS} \mid \mathit{consS}\ \mathit{Char}\ \mathit{String},$$

we can represent string-indexed tries with associated values of type V as follows.

$$\begin{array}{lcl} \textbf{data}\ \mathit{FMapString}\ V & = & \mathit{nullString} \\ & \mid & \mathit{trieString}\ (\mathit{Maybe}\ V)\ (\mathit{FMapChar}\ (\mathit{FMapString}\ V)) \end{array}$$

Here, *nullString* represents the empty trie. The first component of the constructor *trieString* contains the value associated with *nilS*. Its type is *Maybe* V instead of V since *nilS* may not be in the domain of the finite map represented by the trie. In this case the first component equals *nothing*. The second component corresponds to the edge map. To keep the introductory example manageable we implement *FMapChar* using ordered association lists.

$$\begin{array}{lcl} \textbf{type}\ \mathit{FMapChar}\ V & = & [(\mathit{Char}, V)] \\ \mathit{lookupChar} & :: & \forall V\,.\,\mathit{Char} \to \mathit{FMapChar}\ V \to \mathit{Maybe}\ V \\ \mathit{lookupChar}\ c\ [\,] & = & \mathit{nothing} \\ \mathit{lookupChar}\ c\ ((c', v) : x) & & \\ \quad \mid c < c' & = & \mathit{nothing} \\ \quad \mid c == c' & = & \mathit{just}\ v \\ \quad \mid c > c' & = & \mathit{lookupChar}\ c\ x \end{array}$$

Note that *lookupChar* has result type *Maybe* V. If the key is not in the domain of the finite map, *nothing* is returned.

Building upon *lookupChar* we can define a look-up function for strings. To look up the empty string we access the first component of the trie. To look up a non-empty string, say, *consS* c s we look up c in the edge map obtaining a trie, which is then recursively searched for s.

$$\begin{array}{lcl} \mathit{lookupString} & :: & \forall V\,.\,\mathit{String} \to \mathit{FMapString}\ V \to \mathit{Maybe}\ V \\ \mathit{lookupString}\ s\ \mathit{nullString} & = & \mathit{nothing} \\ \mathit{lookupString}\ \mathit{nilS}\ (\mathit{trieString}\ \mathit{tn}\ \mathit{tc}) & = & \mathit{tn} \\ \mathit{lookupString}\ (\mathit{consS}\ c\ s)\ (\mathit{trieString}\ \mathit{tn}\ \mathit{tc}) & = & (\mathit{lookupChar}\ c \diamond \mathit{lookupString}\ s)\ \mathit{tc} \end{array}$$

In the last equation we use monadic composition to take care of the error signal *nothing*.

Based on work by Wadsworth and others, Connelly and Morris (1995) have generalized the concept of a trie to permit indexing by elements built according to an arbitrary signature, that is, by elements of an arbitrary non-parameterized data type. The definition of *lookupString* already gives a clue what a suitable generalization might look like: the trie *trieString tn tc* contains a finite map for each constructor of the data type *String*; to look up *consS* c s the look-up functions for the components, c and s, are composed. Generally, if we have a data type with k constructors, the corresponding trie has k components. To look up a constructor with n fields, we must select the corresponding finite map and compose n look-up functions of the appropriate types. If a constructor has no fields such as *nilS*, we extract the associated value.

As a second example, consider the data type of external search trees (a parametric version of this type was introduced in Section 2.1.2 on page 17):

$$\textbf{data}\ \mathit{Dict} \ = \ \mathit{leaf}\ \mathit{String} \mid \mathit{node}\ \mathit{Dict}\ \mathit{String}\ \mathit{Dict}.$$

A trie for external search trees represents a finite map from *Dict* to some value type V. It is an element of *FMapDict* V given by

$$\begin{array}{lcl} \textbf{data}\ \mathit{FMapDict}\ V & = & \mathit{nullDict} \\ & \mid & \mathit{trieDict}\ (\mathit{FMapString}\ V)\ (\mathit{FMapDict}\ (\mathit{FMapString}\ (\mathit{FMapDict}\ V))). \end{array}$$

Note that *FMapDict* is a nested data type, since the recursive call on the right hand side, *FMapDict* (*FMapString* (*FMapDict* V)), is a substitution instance of the left hand side. Consequently, the look-up function on external search trees requires polymorphic recursion.

$$\begin{array}{lcl}
\mathit{lookupDict} & :: & \forall V\,.\,\mathit{Dict} \to \mathit{FMapDict}\ V \to \mathit{Maybe}\ V \\
\mathit{lookupDict}\ d\ \mathit{nullDict} & = & \mathit{nothing} \\
\mathit{lookupDict}\ (\mathit{leaf}\ s)\ (\mathit{trieDict}\ \mathit{tl}\ \mathit{tn}) & = & \mathit{lookupString}\ s\ \mathit{tl} \\
\mathit{lookupDict}\ (\mathit{node}\ l\ s\ r)\ (\mathit{trieDict}\ \mathit{tl}\ \mathit{tn}) & = & (\mathit{lookupDict}\ l \diamond \mathit{lookupString}\ s \diamond \mathit{lookupDict}\ r)\ \mathit{tn}
\end{array}$$

Looking up a node involves two recursive calls. The first, *lookupDict* l, is of type $\mathit{Dict} \to \mathit{FMapDict}\ X \to \mathit{Maybe}\ X$ where $X = \mathit{FMapString}\ (\mathit{FMapDict}\ V)$, which is a substitution instance of the declared type.

Note that it is absolutely necessary that *FMapDict* and *lookupDict* are parametric with respect to the codomain of the finite maps. Had we restricted the type of *lookupDict* to $\mathit{Dict} \to \mathit{FMapDict}\ T \to T$ for some fixed type T, the definition would have no longer type-checked. This also explains why the construction does not work for the finite set abstraction.

Remark 5.3 Looking up a constructed value boils down to composing look-up functions. Interestingly, the order of composition is completely arbitrary: we are free to use either textual order or reverse textual order. For instance, *FMapString* and *lookupString* can alternatively be defined by

$$\begin{array}{lcl}
\mathbf{data}\ \mathit{FMapString}\ V & = & \mathit{nullString} \\
 & \mid & \mathit{trieString}\ (\mathit{Maybe}\ V)\ (\mathit{FMapString}\ (\mathit{FMapChar}\ V)) \\
\mathit{lookupString} & :: & \forall V\,.\,\mathit{String} \to \mathit{FMapString}\ V \to \mathit{Maybe}\ V \\
\mathit{lookupString}\ s\ \mathit{nullString} & = & \mathit{nothing} \\
\mathit{lookupString}\ \mathit{nilS}\ (\mathit{trieString}\ \mathit{tn}\ \mathit{tc}) & = & \mathit{tn} \\
\mathit{lookupString}\ (\mathit{consS}\ c\ s)\ (\mathit{trieString}\ \mathit{tn}\ \mathit{tc}) & & \\
 & = & (\mathit{lookupString}\ s \diamond \mathit{lookupChar}\ c)\ \mathit{tc}.
\end{array}$$

These definitions employ reverse textual order—s is looked up first and then c—and correspond to the textual order implementation of tries for 'snoc' strings given by **data** $\mathit{Gnirts} = \mathit{lin} \mid \mathit{snoc}\ \mathit{Gnirts}\ \mathit{Char}$. That said, it becomes clear that both orders must work equally well. As an aside, note that *FMapString* is now a nested data type and *lookupString* requires polymorphic recursion. □

Generalized tries make a particularly interesting application of generic programming. The central insight is that a trie can be considered as a *type-indexed data type*. This makes it possible to define tries and operations on tries generically for arbitrary data types. We already have the necessary prerequisites at hand: we know how to define tries for sums and for products. A trie for a sum is essentially a product of tries and a trie for a product is a composition of tries. The extension to arbitrary data types is then uniquely defined. Mathematically speaking, generalized tries are based on the following isomorphisms.

$$\begin{array}{lcl}
1 \to_{\mathrm{fin}} V & \cong & V \\
(K_1 + K_2) \to_{\mathrm{fin}} V & \cong & (K_1 \to_{\mathrm{fin}} V) \times (K_2 \to_{\mathrm{fin}} V) \\
(K_1 \times K_2) \to_{\mathrm{fin}} V & \cong & K_1 \to_{\mathrm{fin}} (K_2 \to_{\mathrm{fin}} V)
\end{array}$$

Here, $K \to_{\mathrm{fin}} V$ denotes the set of all finite maps from K to V. Note that $K \to_{\mathrm{fin}} V$ is sometimes written $V^{[K]}$, which explains why these equations are also known as the 'laws of exponentials', see also Section 2.3.7.

5.5.2 Signature

To put the above idea in concrete terms we will define a scheme for constructing data types

$$FMap\langle K :: \star\rangle \quad :: \quad \star \to \star,$$

which assigns a type constructor of kind $\star \to \star$ to each key type K of kind $\star$.

The type $FMap\langle K\rangle\ V$ represents the set $K \to_{\mathrm{fin}} V$ of finite maps from K to V. It is worth noting that the two arguments of '$\to_{\mathrm{fin}}$' are treated in a different way: the key type K is used as a type index, that is, *FMap* will be defined by induction on the structure of K, whereas V is a type parameter, that is, *FMap* will be parametric in the value type V and the operations on tries will be polymorphic with respect to V.

We will implement the following operations on tries.

$$\begin{array}{lll}
empty\langle K\rangle & :: & \forall V\,.\,FMap\langle K\rangle\ V \\
single\langle K\rangle & :: & \forall V\,.\,K \times V \to FMap\langle K\rangle\ V \\
lookup\langle K\rangle & :: & \forall V\,.\,K \to FMap\langle K\rangle\ V \to Maybe\ V \\
insert\langle K\rangle & :: & \forall V\,.\,(V \to V \to V) \to K \times V \to (FMap\langle K\rangle\ V \to FMap\langle K\rangle\ V) \\
merge\langle K\rangle & :: & \forall V\,.\,(V \to V \to V) \to (FMap\langle K\rangle\ V \to FMap\langle K\rangle\ V \to FMap\langle K\rangle\ V)
\end{array}$$

The value $empty\langle K\rangle$ is the empty trie; $single\langle K\rangle\ (k, v)$ constructs a trie that contains the binding (k, v) as the single element. The function $lookup\langle K\rangle$ takes a key and a trie and looks up the value associated with the key. The function $insert\langle K\rangle$ inserts a new binding into a trie and $merge\langle K\rangle$ combines two tries. The two latter functions take as a first argument a so-called *combining function*, which is applied whenever two bindings have the same key. For instance, $\lambda new\ old \to new$ is used as the combining function for $insert\langle K\rangle$ if the new binding is to override an old binding with the same key. For finite maps of type $FMap\langle K\rangle\ Int$ addition may also be a sensible choice. Interestingly, we will see that the combining function is not only a convenient feature for the user; it is also necessary for defining $insert\langle K\rangle$ and $merge\langle K\rangle$ generically for all types!

5.5.3 Type-indexed tries

We have already noted that generalized tries are based on the laws of exponentials.

$$\begin{array}{lll}
1 \to_{\mathrm{fin}} V & \cong & V \\
(K_1 + K_2) \to_{\mathrm{fin}} V & \cong & (K_1 \to_{\mathrm{fin}} V) \times (K_2 \to_{\mathrm{fin}} V) \\
(K_1 \times K_2) \to_{\mathrm{fin}} V & \cong & K_1 \to_{\mathrm{fin}} (K_2 \to_{\mathrm{fin}} V)
\end{array}$$

In order to define the notion of finite map it is customary to assume that each value type V contains a distinguished element or *base point* $\perp_V$, see Connelly and Morris (1995). A finite map is then a function whose value is $\perp_V$ for all but finitely many arguments. For the implementation of tries it is, however, inconvenient to make such a strong assumption (though one could use type classes for this purpose). Instead, we explicitly add a base point when necessary motivating the following definition of *FMap*:

$$\begin{array}{lll}
FMap\langle K :: \star\rangle & :: & \star \to \star \\
FMap\langle 1\rangle & = & \Lambda V\,.\,Maybe\ V \\
FMap\langle Char\rangle & = & \Lambda V\,.\,FMapChar\ V \\
FMap\langle Int\rangle & = & \Lambda V\,.\,Patricia.Dict\ V \\
FMap\langle K_1 + K_2\rangle & = & \Lambda V\,.\,FMap\langle K_1\rangle\ V \times_{\bullet} FMap\langle K_2\rangle\ V \\
FMap\langle K_1 \times K_2\rangle & = & \Lambda V\,.\,FMap\langle K_1\rangle\ (FMap\langle K_2\rangle\ V).
\end{array}$$

Here, $(\times_\bullet)$ is the type of optional pairs (see Section 2.1.1).

$$\textbf{data}\ A \times_\bullet B \quad = \quad null \mid pair\ A\ B.$$

We take for granted the existence of a suitable library implementing finite maps with integer keys. Such a library could be based, for instance, on a data structure known as a *Patricia tree* (Okasaki and Gill 1998). This data structure fits particularly well in the current setting since Patricia trees are a variety of tries. For clarity, we will use qualified names when referring to entities defined in the hypothetical module *Patricia*.

Note that $FMap\langle K\rangle$ is a unary functor. Using functorial notation we can define *FMap* more succinctly as

$$\begin{array}{lcl} FMap\langle 1\rangle & = & Maybe \\ FMap\langle Char\rangle & = & FMapChar \\ FMap\langle Int\rangle & = & Patricia.Dict \\ FMap\langle K_1 + K_2\rangle & = & FMap\langle K_1\rangle \mathbin{\underline{\times}_\bullet} FMap\langle K_2\rangle \\ FMap\langle K_1 \times k_2\rangle & = & FMap\langle K_1\rangle \cdot FMap\langle K_2\rangle. \end{array}$$

We will show that a trie is a functor for a slight variation of *FMap* in Section 5.6.6.

Since the trie for the unit type is given by *Maybe* rather than *Id*, tries for isomorphic types are, in general, not isomorphic. We have, for instance, $1 \cong 1 \times 1$ but $FMap\langle 1\rangle = Maybe \not\cong Maybe \cdot Maybe = FMap\langle 1 \times 1\rangle$. The trie type $Maybe \cdot Maybe$ has two different representations of the empty trie: *nothing* and *just nothing*. However, only the first one will be used in our implementation. Similarly, $Maybe \mathbin{\underline{\times}_\bullet} Maybe$ has two elements, *null* and *pair nothing nothing*, that represent the empty trie. Again, only the first one will be used.

REMARK 5.4 Instead of optional pairs we can also use ordinary pairs in the definition of *FMap*:

$$FMap\langle K_1 + K_2\rangle \quad = \quad \Lambda V\,.\,FMap\langle K_1\rangle\ V \times FMap\langle K_2\rangle\ V.$$

This representation has, however, two major drawbacks: (i) it relies in an essential way on lazy evaluation and (ii) it is inefficient, see Hinze (2000b). □

Building upon the techniques developed in Section 3.1.3 we can now specialize $FMap\langle T\rangle$ for a given instance of T. That is, for each type constructor T of kind $\mathfrak{T}$ we define a *higher-order type constructor* $FMap\langle\!\langle T\rangle\!\rangle$. For $\mathfrak{T} = \star \to \star$ we have, for instance,

$$FMap\langle\!\langle F :: \star \to \star\rangle\!\rangle \quad :: \quad (\star \to \star) \to (\star \to \star).$$

The type constructor $FMap\langle\!\langle F\rangle\!\rangle$ is *the generalized trie of the unary type constructor* F. It takes as argument the generalized trie of the base type, say, A and yields the generalized trie of $F\ A$. It may come as a surprise that the framework for specializing type-indexed values is also applicable to type-indexed data types. The following equations show how to extend *FMap* to arbitrary type terms of arbitrary kinds.

$$
\begin{array}{lcl}
\mathfrak{FMap}\langle \mathfrak{T} :: \Box \rangle & :: & \Box \\
\mathfrak{FMap}\langle \star \rangle & = & \star \to \star \\
\mathfrak{FMap}\langle \mathfrak{A} \times \mathfrak{B} \rangle & = & \mathfrak{FMap}\langle \mathfrak{A} \rangle \times \mathfrak{FMap}\langle \mathfrak{B} \rangle \\
\mathfrak{FMap}\langle \mathfrak{A} \to \mathfrak{B} \rangle & = & \mathfrak{FMap}\langle \mathfrak{A} \rangle \to \mathfrak{FMap}\langle \mathfrak{B} \rangle \\
\mathit{FMap}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathfrak{FMap}\langle \mathfrak{T} \rangle\ T \\
\mathit{FMap}\langle\!\langle A \rangle\!\rangle & = & \mathit{FMap}_A \\
\mathit{FMap}\langle\!\langle (T_1, T_2) \rangle\!\rangle & = & (\mathit{FMap}\langle\!\langle T_1 \rangle\!\rangle, \mathit{FMap}\langle\!\langle T_2 \rangle\!\rangle) \\
\mathit{FMap}\langle\!\langle \mathit{Outl}\ T \rangle\!\rangle & = & \mathit{Outl}\ (\mathit{FMap}\langle\!\langle T \rangle\!\rangle) \\
\mathit{FMap}\langle\!\langle \mathit{Outr}\ T \rangle\!\rangle & = & \mathit{Outr}\ (\mathit{FMap}\langle\!\langle T \rangle\!\rangle) \\
\mathit{FMap}\langle\!\langle \Lambda A \,.\, T \rangle\!\rangle & = & \Lambda \mathit{FMap}_A \,.\, \mathit{FMap}\langle\!\langle T \rangle\!\rangle \\
\mathit{FMap}\langle\!\langle T\ U \rangle\!\rangle & = & (\mathit{FMap}\langle\!\langle T \rangle\!\rangle)\ (\mathit{FMap}\langle\!\langle U \rangle\!\rangle) \\
\mathit{FMap}\langle\!\langle \mathit{Fix}\ T \rangle\!\rangle & = & \mathit{Fix}\ (\mathit{FMap}\langle\!\langle T \rangle\!\rangle)
\end{array}
$$

Note that the kind of $\mathit{FMap}\langle\!\langle T \rangle\!\rangle$ depends on the kind of T. Consequently, $\mathfrak{FMap}$ is a kind-indexed kind.

Example 5.5 Let us specialize *FMap* to the following data types introduced in Sections 1.1 and 2.1.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{List}\ A & = & \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A) \\
\textbf{data}\ \mathit{Tree}\ A\ B & = & \mathit{leaf}\ A \mid \mathit{node}\ (\mathit{Tree}\ A\ B)\ B\ (\mathit{Tree}\ A\ B) \\
\textbf{data}\ \mathit{Fork}\ A & = & \mathit{fork}\ A\ A \\
\textbf{data}\ \mathit{Sequ}\ A & = & \mathit{endS} \mid \mathit{zeroS}\ (\mathit{Sequ}\ (\mathit{Fork}\ A)) \mid \mathit{oneS}\ A\ (\mathit{Sequ}\ (\mathit{Fork}\ A))
\end{array}
$$

Recall that these types are represented by

$$
\begin{array}{lcl}
\mathit{List} & = & \mathit{Fix}\ (\Lambda \mathit{List} \,.\, \Lambda A \,.\, 1 + A \times \mathit{List}\ A) \\
\mathit{Tree} & = & \mathit{Fix}\ (\Lambda \mathit{Tree} \,.\, \Lambda A\ B \,.\, A + \mathit{Tree}\ A\ B \times B \times \mathit{Tree}\ A\ B) \\
\mathit{Fork} & = & \Lambda A \,.\, A \times A \\
\mathit{Sequ} & = & \mathit{Fix}\ (\Lambda \mathit{Sequ} \,.\, \Lambda A \,.\, 1 + \mathit{Sequ}\ (\mathit{Fork}\ A) + A \times \mathit{Sequ}\ (\mathit{Fork}\ A)).
\end{array}
$$

Consequently, the corresponding trie types are

$$
\begin{array}{lcl}
\mathit{FMapList} & = & \mathit{Fix}\ (\Lambda \mathit{FMapList} \,.\, \Lambda \mathit{FA} \,.\, \mathit{Maybe} \times_\bullet \mathit{FA} \cdot \mathit{FMapList}\ \mathit{FA}) \\
\mathit{FMapTree} & = & \mathit{Fix}\ (\Lambda \mathit{FMapTree} \,.\, \Lambda \mathit{FA}\ \mathit{FB} \,. \\
& & \quad \mathit{FA} \times_\bullet \\
& & \quad \mathit{FMapTree}\ \mathit{FA}\ \mathit{FB} \cdot \mathit{FB} \cdot \mathit{FMapTree}\ \mathit{FA}\ \mathit{FB}) \\
\mathit{FMapFork} & = & \Lambda \mathit{FA} \,.\, \mathit{FA} \cdot \mathit{FA} \\
\mathit{FMapSequ} & = & \mathit{Fix}\ (\Lambda \mathit{FMapSequ} \,.\, \Lambda \mathit{FA} \,. \\
& & \quad \mathit{Maybe} \times_\bullet \\
& & \quad \mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA}) \times_\bullet \\
& & \quad \mathit{FA} \cdot \mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})).
\end{array}
$$

As an aside, note that we interpret $A_1 \times_\bullet A_2 \times_\bullet A_3$ as the type of optional triples and not as nested optional pairs:

$$\textbf{data}\ A_1 \times_\bullet A_2 \times_\bullet A_3 \quad = \quad \mathit{null} \mid \mathit{triple}\ A_1\ A_2\ A_3.$$

Now, since Haskell permits the definition of higher-order kinded data types, the second-order type constructors above can be directly coded as data types. All we have to do is to bring the equations into an applicative form.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{FMapList}\ \mathit{FA}\ V & = & \mathit{nullList} \\
 & \mid & \mathit{trieList}\ (\mathit{Maybe}\ V) \\
 & & \qquad (\mathit{FA}\ (\mathit{FMapList}\ \mathit{FA}\ V)) \\
\textbf{data}\ \mathit{FMapTree}\ \mathit{FA}\ \mathit{FB}\ V & = & \mathit{nullTree} \\
 & \mid & \mathit{trieTree}\ (\mathit{FA}\ V) \\
 & & \qquad (\mathit{FMapTree}\ \mathit{FA}\ \mathit{FB} \\
 & & \qquad\quad (\mathit{FB}\ (\mathit{FMapTree}\ \mathit{FA}\ \mathit{FB}\ V)))
\end{array}
$$

These types are the parametric variants of *FMapString* and *FMapDict* defined in Section 5.5.1: we have $\mathit{FMapString} \approx \mathit{FMapList}\ \mathit{FMapChar}$ (corresponding to $\mathit{String} \approx \mathit{List}\ \mathit{Char}$) and $\mathit{FMapDict} \approx \mathit{FMapTree}\ \mathit{FMapString}\ \mathit{FMapString}$ (corresponding to $\mathit{Dict} \approx \mathit{Tree}\ \mathit{String}\ \mathit{String}$). Things become interesting if we consider nested data types.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{FMapFork}\ \mathit{FA}\ V & = & \mathit{trieFork}\ (\mathit{FA}\ (\mathit{FA}\ V)) \\
\textbf{data}\ \mathit{FMapSequ}\ \mathit{FA}\ V & = & \mathit{nullSequ} \\
 & \mid & \mathit{trieSequ}\ (\mathit{Maybe}\ V) \\
 & & \qquad (\mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})\ V) \\
 & & \qquad (\mathit{FA}\ (\mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})\ V))
\end{array}
$$

The generalized trie of a nested data type is a second-order nested data type! A nest is termed second-order, if a parameter that is instantiated in a recursive call ranges over type constructors of first-order kind. The trie *FMapSequ* is a second-order nest since the parameter *FA* of kind $\star \to \star$ is changed in the recursive calls. By contrast, *FMapTree* is a first-order nest since its instantiated parameter V has kind $\star$. It is quite easy to produce generalized tries that are both first- and second-order nests. If we swap the components of *Sequ*'s third constructor—$\mathit{oneS}\ a\ (\mathit{Sequ}\ (\mathit{Fork}\ a))$ becomes $\mathit{oneS}\ (\mathit{Sequ}\ (\mathit{Fork}\ a))\ a$—then the third component of *FMapSequ* has type $\mathit{FMapSequ}\ (\mathit{FMapFork}\ \mathit{FA})\ (\mathit{FA}\ V)$ and since both *FA* and V are instantiated, *FMapSequ* is consequently both a first- and a second-order nest. □

5.5.4 Empty tries

The empty trie is defined as follows.

$$
\begin{array}{lcl}
\mathit{empty}\langle K\rangle & :: & \forall V\,.\,\mathit{FMap}\langle K\rangle\ V \\
\mathit{empty}\langle 1\rangle & = & \mathit{nothing} \\
\mathit{empty}\langle \mathit{Char}\rangle & = & [\,] \\
\mathit{empty}\langle \mathit{Int}\rangle & = & \mathit{Patricia.empty} \\
\mathit{empty}\langle K_1 + K_2\rangle & = & \mathit{null} \\
\mathit{empty}\langle K_1 \times K_2\rangle & = & \mathit{empty}\langle K_1\rangle
\end{array}
$$

The definition already illustrates several interesting aspects of programming with generalized tries. To begin with the polymorphic type of *empty* is necessary to make the definition work. Consider the last equation: $\mathit{empty}\langle K_1 \times K_2\rangle$, which is of type $\forall V\,.\,\mathit{FMap}\langle K_1\rangle\ (\mathit{FMap}\langle K_2\rangle\ V)$, is defined in terms of $\mathit{empty}\langle K_1\rangle$, which is of type $\forall V\,.\,\mathit{FMap}\langle K_1\rangle\ V$. That means that $\mathit{empty}\langle K_1\rangle$ is used polymorphically. In other words, *empty* makes use of polymorphic recursion!

The specialization of *empty* works essentially as before. Applying the scheme of Section 3.1.3 we obtain

$$
\begin{array}{lcl}
\mathit{Empty}\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{T} \to \star \\
\mathit{Empty}\langle \star \rangle\ T & = & \forall V \,.\, \mathit{FMap}\langle T \rangle\ V \\
\mathit{Empty}\langle \mathfrak{A} \times \mathfrak{B} \rangle\ T & = & \mathit{Empty}\langle \mathfrak{A} \rangle\ (\mathit{Outl}\ T) \times \mathit{Empty}\langle \mathfrak{B} \rangle\ (\mathit{Outr}\ T) \\
\mathit{Empty}\langle \mathfrak{A} \to \mathfrak{B} \rangle\ T & = & \forall A \,.\, \mathit{Empty}\langle \mathfrak{A} \rangle\ A \to \mathit{Empty}\langle \mathfrak{B} \rangle\ (T\ A) \\
\mathit{empty}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle & :: & \mathit{Empty}\langle \mathfrak{T} \rangle\ T \\
\mathit{empty}\langle\!\langle C \rangle\!\rangle & = & \mathit{empty}_C \\
\mathit{empty}\langle\!\langle A \rangle\!\rangle & = & \mathit{empty}_A \\
\mathit{empty}\langle\!\langle (T_1, T_2) \rangle\!\rangle & = & (\mathit{empty}\langle\!\langle T_1 \rangle\!\rangle, \mathit{empty}\langle\!\langle T_2 \rangle\!\rangle) \\
\mathit{empty}\langle\!\langle \mathit{Outl}\ T \rangle\!\rangle & = & \mathit{outl}\ (\mathit{empty}\langle\!\langle T \rangle\!\rangle) \\
\mathit{empty}\langle\!\langle \mathit{Outr}\ T \rangle\!\rangle & = & \mathit{outr}\ (\mathit{empty}\langle\!\langle T \rangle\!\rangle) \\
\mathit{empty}\langle\!\langle \Lambda A \,.\, T \rangle\!\rangle & = & \lambda \mathit{empty}_A \,.\, \mathit{empty}\langle\!\langle T \rangle\!\rangle \\
\mathit{empty}\langle\!\langle T\ U \rangle\!\rangle & = & (\mathit{empty}\langle\!\langle T \rangle\!\rangle)\ (\mathit{empty}\langle\!\langle U \rangle\!\rangle) \\
\mathit{empty}\langle\!\langle \mathit{Fix}\ T \rangle\!\rangle & = & \mathit{fix}\ (\mathit{empty}\langle\!\langle T \rangle\!\rangle).
\end{array}
$$

There is one small glitch, however. Consider the type signature of $\mathit{empty}\langle\!\langle F \rangle\!\rangle$ where F is a type constructor of kind $\star \to \star$.

$$\mathit{empty}\langle\!\langle F \rangle\!\rangle \quad :: \quad \forall A \,.\, (\forall W \,.\, \mathit{FMap}\langle A \rangle\ W) \to (\forall V \,.\, \mathit{FMap}\langle F\ A \rangle\ V)$$

The type signature contains two occurrences of FMap. Of course, if we want to specialize empty for a given F, we must specialize its type signature, as well. To this end we replace $\mathit{FMap}\langle F\ A \rangle$ by $\mathit{FMap}\langle F \rangle\ (\mathit{FMap}\langle A \rangle)$ and generalize $\mathit{FMap}\langle A \rangle$ to a fresh type variable, say, FA.

$$\mathit{empty}\langle\!\langle F \rangle\!\rangle \quad :: \quad \forall \mathit{FA} \,.\, (\forall W \,.\, \mathit{FA}\ W) \to (\forall V \,.\, \mathit{FMap}\langle F \rangle\ \mathit{FA}\ V)$$

The following refined definition of Empty captures this generalization.

$$
\begin{array}{lcl}
\mathit{Empty}\langle \mathfrak{T} :: \Box \rangle & :: & \mathfrak{FMap}\langle \mathfrak{T} \rangle \to \star \\
\mathit{Empty}\langle \star \rangle\ \mathit{FT} & = & \forall V \,.\, \mathit{FT}\ V \\
\mathit{Empty}\langle \mathfrak{A} \times \mathfrak{B} \rangle\ \mathit{FT} & = & \mathit{Empty}\langle \mathfrak{A} \rangle\ (\mathit{Outl}\ \mathit{FT}) \times \mathit{Empty}\langle \mathfrak{B} \rangle\ (\mathit{Outr}\ \mathit{FT}) \\
\mathit{Empty}\langle \mathfrak{A} \to \mathfrak{B} \rangle\ \mathit{FT} & = & \forall \mathit{FA} \,.\, \mathit{Empty}\langle \mathfrak{A} \rangle\ \mathit{FA} \to \mathit{Empty}\langle \mathfrak{B} \rangle\ (\mathit{FT}\ \mathit{FA})
\end{array}
$$

It is not hard to see that $\mathit{Empty}\langle \mathfrak{T} \rangle\ (\mathit{FMap}\langle T \rangle)$ is a valid type of $\mathit{empty}\langle\!\langle T :: \mathfrak{T} \rangle\!\rangle$.

Example 5.6 Let us specialize *empty* to lists and binary random-access lists.

$$
\begin{array}{lcl}
\mathit{emptyList} & :: & \forall \mathit{FA} \,.\, (\forall W \,.\, \mathit{FA}\ W) \to (\forall V \,.\, \mathit{FMapList}\ \mathit{FA}\ V) \\
\mathit{emptyList}\ \mathit{eA} & = & \mathit{nullList} \\
\mathit{emptyFork} & :: & \forall \mathit{FA} \,.\, (\forall W \,.\, \mathit{FA}\ W) \to (\forall V \,.\, \mathit{FMapFork}\ \mathit{FA}\ V) \\
\mathit{emptyFork}\ \mathit{eA} & = & \mathit{trieFork}\ \mathit{eA} \\
\mathit{emptySequ} & :: & \forall \mathit{FA} \,.\, (\forall W \,.\, \mathit{FA}\ W) \to (\forall V \,.\, \mathit{FMapSequ}\ \mathit{FA}\ V) \\
\mathit{emptySequ}\ \mathit{eA} & = & \mathit{nullSequ}
\end{array}
$$

The second function, *emptyFork*, illustrates the polymorphic use of the parameter: eA has type $\forall W \,.\, \mathit{FA}\ W$ but is used as an element of $\mathit{FA}\ (\mathit{FA}\ W)$. The functions *emptyList* and *emptySequ* show that the 'mechanically' generated definitions can sometimes be slightly improved: the argument eA is not needed. □

5.5.5 Singleton tries

The singleton trie, which contains only a single binding, is defined as follows.

$$
\begin{array}{lcl}
single\langle K\rangle & :: & \forall V\,.\,K \times V \to FMap\langle K\rangle\ V\\
single\langle 1\rangle\ ((), v) & = & just\ v\\
single\langle Char\rangle\ (k, v) & = & [(k, v)]\\
single\langle Int\rangle\ (k, v) & = & Patricia.single\ (k, v)\\
single\langle K_1 + K_2\rangle\ (inl\ k_1, v) & = & pair\ (single\langle K_1\rangle\ (k_1, v))\ (empty\langle K_2\rangle)\\
single\langle K_1 + K_2\rangle\ (inr\ k_2, v) & = & pair\ (empty\langle K_1\rangle)\ (single\langle K_2\rangle\ (k_2, v))\\
single\langle K_1 \times K_2\rangle\ ((k_1, k_2), v) & = & single\langle K_1\rangle\ (k_1, single\langle K_2\rangle\ (k_2, v))
\end{array}
$$

The definition of *single* is interesting because it falls back on *empty* in the fourth and the fifth equation. This requires a small extension of the theory: the specialization must be parameterized both with *single* and with *empty*. In fact, *empty* and *single* can be seen as being defined by mutual recursion (ignoring the fact that *empty* does not call *single*).

EXAMPLE 5.7 Let us again specialize the generic function to lists and binary random-access lists.

$$
\begin{array}{lcl}
singleList & :: & \forall K\ FA\,.\,(\forall W\,.\,FA\ W) \to (\forall W\,.\,K \times W \to FA\ W)\\
 & & \quad \to (\forall V\,.\,(List\ K \times V \to FMapList\ FA\ V))\\
singleList\ eA\ sA\ (nil, v) & = & trieList\ (just\ v)\ eA\\
singleList\ eA\ sA\ (cons\ k\ ks, v) & = & trieList\ nothing\ (sA\ (k, singleList\ eA\ sA\ (ks, v)))\\
singleFork & :: & \forall K\ FA\,.\,(\forall W\,.\,FA\ W) \to (\forall W\,.\,K \times W \to FA\ W)\\
 & & \quad \to (\forall V\,.\,(Fork\ K \times V \to FMapFork\ FA\ V))\\
singleFork\ eA\ sA\ (fork\ k_1\ k_2, v) & = & trieFork\ (sA\ (k_1, sA\ (k_2, v)))\\
singleSequ & :: & \forall K\ FA\,.\,(\forall W\,.\,FA\ W) \to (\forall W\,.\,K \times W \to FA\ W)\\
 & & \quad \to (\forall V\,.\,(Sequ\ K \times V \to FMapSequ\ FA\ V))\\
singleSequ\ eA\ sA\ (endS, v) & = & trieSequ\ (just\ v)\ nullSequ\ eA
\end{array}
$$

$$
\begin{array}{l}
singleSequ\ eA\ sA\ (zeroS\ s, v)\\
\quad = \ trieSequ\ nothing\ (singleSequ\ (emptyFork\ eA)\ (singleFork\ eA\ sA)\ (s, v))\ eA\\
singleSequ\ eA\ sA\ (oneS\ k\ s, v)\\
\quad = \ trieSequ\ nothing\ nullSequ\ (sA\ (k, singleSequ\ (emptyFork\ eA)\ (singleFork\ eA\ sA)\ (s, v)))
\end{array}
$$

Again, we can simplify the 'mechanically' generated definitions: since the definition of *Fork* does not involve sums, *singleFork* does not require its first argument, *eA*, which can be safely removed. □

5.5.6 Look up

The look-up function implements the scheme discussed in Section 5.5.1.

$$
\begin{array}{lcl}
lookup\langle K\rangle & :: & \forall V\,.\,K \to FMap\langle K\rangle\ V \to Maybe\ V\\
lookup\langle 1\rangle\ ()\ t & = & t\\
lookup\langle Char\rangle\ k\ t & = & lookupChar\ k\ t\\
lookup\langle Int\rangle\ k\ t & = & Patricia.lookup\ k\ t\\
lookup\langle K_1 + K_2\rangle\ k\ null & = & nothing\\
lookup\langle K_1 + K_2\rangle\ (inl\ k_1)\ (pair\ t_1\ t_2) & = & lookup\langle K_1\rangle\ k_1\ t_1\\
lookup\langle K_1 + K_2\rangle\ (inr\ k_2)\ (pair\ t_1\ t_2) & = & lookup\langle K_2\rangle\ k_2\ t_2\\
lookup\langle K_1 \times K_2\rangle\ (k_1, k_2)\ t_1 & = & (lookup\langle K_1\rangle\ k_1 \diamond lookup\langle K_2\rangle\ k_2)\ t_1
\end{array}
$$

On sums the look-up function selects the appropriate map; on products it 'composes' the look-up functions for the components. Since *lookup* has result type *Maybe v*, we use the monadic composition.

EXAMPLE 5.8 Specializing $lookup\langle K\rangle$ to concrete instances of K is by now probably a matter of routine. We obtain

$$
\begin{array}{lcl}
\mathit{lookupList} :: \forall K\ \mathit{FA}\,.\,(\forall W\,.\,K \to \mathit{FA}\ W \to \mathit{Maybe}\ W) & & \\
\qquad\qquad\qquad \to (\forall V\,.\,\mathit{List}\ K \to \mathit{FMapList}\ \mathit{FA}\ V \to \mathit{Maybe}\ V) & & \\
\mathit{lookupList}\ \mathit{lA}\ \mathit{ks}\ \mathit{nullList} & = & \mathit{nothing} \\
\mathit{lookupList}\ \mathit{lA}\ \mathit{nil}\ (\mathit{trieList}\ \mathit{tn}\ \mathit{tc}) & = & \mathit{tn} \\
\mathit{lookupList}\ \mathit{lA}\ (\mathit{cons}\ k\ \mathit{ks})\ (\mathit{trieList}\ \mathit{tn}\ \mathit{tc}) & = & (\mathit{lA}\ k \diamond \mathit{lookupList}\ \mathit{lA}\ \mathit{ks})\ \mathit{tc} \\
\mathit{lookupFork} :: \forall K\ \mathit{FA}\,.\,(\forall W\,.\,K \to \mathit{FA}\ W \to \mathit{Maybe}\ W) & & \\
\qquad\qquad\qquad \to (\forall V\,.\,\mathit{Fork}\ K \to \mathit{FMapFork}\ \mathit{FA}\ V \to \mathit{Maybe}\ V) & & \\
\mathit{lookupFork}\ \mathit{lA}\ (\mathit{fork}\ k_1\ k_2)\ (\mathit{trieFork}\ \mathit{tf}) & = & (\mathit{lA}\ k_1 \diamond \mathit{lA}\ k_2)\ \mathit{tf} \\
\mathit{lookupSequ} :: \forall \mathit{FA}\ K\,.\,(\forall W\,.\,K \to \mathit{FA}\ W \to \mathit{Maybe}\ W) & & \\
\qquad\qquad\qquad \to (\forall V\,.\,\mathit{Sequ}\ K \to \mathit{FMapSequ}\ \mathit{FA}\ V \to \mathit{Maybe}\ V) & & \\
\mathit{lookupSequ}\ \mathit{lA}\ s\ \mathit{nullSequ} & = & \mathit{nothing} \\
\mathit{lookupSequ}\ \mathit{lA}\ \mathit{endS}\ (\mathit{trieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) & = & \mathit{te} \\
\mathit{lookupSequ}\ \mathit{lA}\ (\mathit{zeroS}\ s)\ (\mathit{trieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) & = & \mathit{lookupSequ}\ (\mathit{lookupFork}\ \mathit{lA})\ s\ \mathit{tz} \\
\mathit{lookupSequ}\ \mathit{lA}\ (\mathit{oneS}\ a\ s)\ (\mathit{trieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to}) & = & (\mathit{lA}\ a \diamond \mathit{lookupSequ}\ (\mathit{lookupFork}\ \mathit{lA})\ s)\ \mathit{to}
\end{array}
$$

The function *lookupList* generalizes *lookupString* defined in Section 5.5.1; we have $\mathit{lookupString} \approx \mathit{lookupList}\ \mathit{lookupChar}$. □

5.5.7 Inserting and merging

Insertion is defined in terms of *merge* and *single*.

$$
\begin{array}{lcl}
\mathit{insert}\langle K\rangle & :: & \forall V\,.\,(V \to V \to V) \to K \times V \to (\mathit{FMap}\langle K\rangle\ V \to \mathit{FMap}\langle K\rangle\ V) \\
\mathit{insert}\langle K\rangle\ c\ (k, v)\ t & = & \mathit{merge}\langle K\rangle\ c\ (\mathit{single}\langle K\rangle\ (k, v))\ t
\end{array}
$$

Merging two tries is surprisingly simple. Given an auxiliary function for combining two values of type *Maybe*

$$
\begin{array}{lcl}
\mathit{combine} & :: & \forall V\,.\,(V \to V \to V) \to (\mathit{Maybe}\ V \to \mathit{Maybe}\ V \to \mathit{Maybe}\ V) \\
\mathit{combine}\ c\ \mathit{nothing}\ \mathit{nothing} & = & \mathit{nothing} \\
\mathit{combine}\ c\ \mathit{nothing}\ (\mathit{just}\ v_2) & = & \mathit{just}\ v_2 \\
\mathit{combine}\ c\ (\mathit{just}\ v_1)\ \mathit{nothing} & = & \mathit{just}\ v_1 \\
\mathit{combine}\ c\ (\mathit{just}\ v_1)\ (\mathit{just}\ v_2) & = & \mathit{just}\ (c\ v_1\ v_2)
\end{array}
$$

and a function for merging two association lists

$$
\begin{array}{lcl}
\mathit{mergeChar} & :: & \forall V\,.\,(V \to V \to V) \\
 & & \qquad \to (\mathit{FMapChar}\ V \to \mathit{FMapChar}\ V \to \mathit{FMapChar}\ V) \\
\mathit{mergeChar}\ c\ [\,]\ x' & = & x' \\
\mathit{mergeChar}\ c\ x\ [\,] & = & x \\
\mathit{mergeChar}\ c\ ((k, v) : x)\ ((k', v') : x') & & \\
\quad \mid k < k' & = & (k, v) : \mathit{mergeChar}\ c\ x\ ((k', v') : x') \\
\quad \mid k == k' & = & (k, c\ v\ v') : \mathit{mergeChar}\ c\ x\ x' \\
\quad \mid k > k' & = & (k', v') : \mathit{mergeChar}\ c\ ((k, v) : x)\ x',
\end{array}
$$

we can define *merge* as follows.

$$
\begin{array}{lcl}
\mathit{merge}\langle K\rangle & :: & \forall V\,.\,(V \to V \to V) \\
 & & \quad \to (\mathit{FMap}\langle K\rangle\ V \to \mathit{FMap}\langle K\rangle\ V \to \mathit{FMap}\langle K\rangle\ V) \\
\mathit{merge}\langle 1\rangle\ c\ t\ t' & = & \mathit{combine}\ c\ t\ t' \\
\mathit{merge}\langle \mathit{Char}\rangle\ c\ t\ t' & = & \mathit{mergeChar}\ c\ t\ t' \\
\mathit{merge}\langle \mathit{Int}\rangle\ c\ t\ t' & = & \mathit{Patricia.merge}\ c\ t\ t' \\
\mathit{merge}\langle K_1 + K_2\rangle\ c\ \mathit{null}\ t' & = & t' \\
\mathit{merge}\langle K_1 + K_2\rangle\ c\ t\ \mathit{null} & = & t \\
\mathit{merge}\langle K_1 + K_2\rangle\ c\ (\mathit{pair}\ t_1\ t_2)\ (\mathit{pair}\ t_1'\ t_2') & & \\
 & = & \mathit{pair}\ (\mathit{merge}\langle K_1\rangle\ c\ t_1\ t_1')\ (\mathit{merge}\langle K_2\rangle\ c\ t_2\ t_2') \\
\mathit{merge}\langle K_1 \times K_2\rangle\ c\ t\ t' & = & \mathit{merge}\langle K_1\rangle\ (\mathit{merge}\langle K_2\rangle\ c)\ t\ t'
\end{array}
$$

The most interesting equation is the last one. The tries t and t' are of type $\mathit{FMap}\langle K_1 \times K_2\rangle\ V = \mathit{FMap}\langle K_1\rangle\ (\mathit{FMap}\langle K_2\rangle\ V)$. To merge them we can recursively call $\mathit{merge}\langle K_1\rangle$; we must, however, supply a combining function of type $\forall V\,.\,\mathit{FMap}\langle K_2\rangle\ V \to \mathit{FMap}\langle K_2\rangle\ V \to \mathit{FMap}\langle K_2\rangle\ V$. A moment's reflection reveals that $\mathit{merge}\langle K_2\rangle\ c$ is the desired combining function. Using functional composition we can write the last equation quite succinctly as

$$\mathit{merge}\langle K_1 \times K_2\rangle \quad = \quad \mathit{merge}\langle K_1\rangle \cdot \mathit{merge}\langle K_2\rangle.$$

The definition of $\mathit{merge}\langle K\rangle$ shows that it is sometimes necessary to implement operations more general than immediately needed. If $\mathit{merge}\langle K\rangle$ had the simplified type $\forall V\,.\,\mathit{FMap}\langle K\rangle\ V \to \mathit{FMap}\langle K\rangle\ V \to \mathit{FMap}\langle K\rangle\ V$, then we would not be able to give a defining equation for $K = K_1 \times K_2$.

EXAMPLE 5.9 To complete the picture let us again specialize the merging operation for lists and binary random-access lists. The different instances of *merge* are surprisingly concise (only the types look complicated).

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{mergeList} :: \forall \mathit{FA}\,.\,(\forall W\,.\,(W \to W \to W) \to (\mathit{FA}\ W \to \mathit{FA}\ W \to \mathit{FA}\ W))} \\
\multicolumn{3}{l}{\qquad\qquad \to (\forall V\,.\,(V \to V \to V)} \\
\multicolumn{3}{l}{\qquad\qquad\quad \to (\mathit{FMapList}\ \mathit{FA}\ V \to \mathit{FMapList}\ \mathit{FA}\ V \to \mathit{FMapList}\ \mathit{FA}\ V))} \\
\mathit{mergeList}\ \mathit{mA}\ c\ \mathit{nullList}\ t & = & t \\
\mathit{mergeList}\ \mathit{mA}\ c\ t\ \mathit{nullList} & = & t \\
\multicolumn{3}{l}{\mathit{mergeList}\ \mathit{mA}\ c\ (\mathit{trieList}\ \mathit{tn}\ \mathit{tc})\ (\mathit{trieList}\ \mathit{tn}'\ \mathit{tc}')} \\
 & = & \mathit{trieList}\ (\mathit{combine}\ c\ \mathit{tn}\ \mathit{tn}')\ (\mathit{mA}\ (\mathit{mergeList}\ \mathit{mA}\ c)\ \mathit{tc}\ \mathit{tc}') \\
\multicolumn{3}{l}{\mathit{mergeFork} :: \forall \mathit{FA}\,.\,(\forall W\,.\,(W \to W \to W) \to (\mathit{FA}\ W \to \mathit{FA}\ W \to \mathit{FA}\ W))} \\
\multicolumn{3}{l}{\qquad\qquad \to (\forall V\,.\,(V \to V \to V)} \\
\multicolumn{3}{l}{\qquad\qquad\quad \to (\mathit{FMapFork}\ \mathit{FA}\ V \to \mathit{FMapFork}\ \mathit{FA}\ V \to \mathit{FMapFork}\ \mathit{FA}\ V))} \\
\multicolumn{3}{l}{\mathit{mergeFork}\ \mathit{mA}\ c\ (\mathit{trieFork}\ \mathit{tf})\ (\mathit{trieFork}\ \mathit{tf}')} \\
 & = & \mathit{trieFork}\ (\mathit{mA}\ (\mathit{mA}\ c)\ \mathit{tf}\ \mathit{tf}') \\
\multicolumn{3}{l}{\mathit{mergeSequ} :: \forall \mathit{FA}\,.\,(\forall W\,.\,(W \to W \to W) \to (\mathit{FA}\ W \to \mathit{FA}\ W \to \mathit{FA}\ W))} \\
\multicolumn{3}{l}{\qquad\qquad \to (\forall V\,.\,(V \to V \to V)} \\
\multicolumn{3}{l}{\qquad\qquad\quad \to (\mathit{FMapSequ}\ \mathit{FA}\ V \to \mathit{FMapSequ}\ \mathit{FA}\ V \to \mathit{FMapSequ}\ \mathit{FA}\ V))} \\
\mathit{mergeSequ}\ \mathit{mA}\ c\ \mathit{nullSequ}\ t & = & t \\
\mathit{mergeSequ}\ \mathit{mA}\ c\ t\ \mathit{nullSequ} & = & t \\
\multicolumn{3}{l}{\mathit{mergeSequ}\ \mathit{mA}\ c\ (\mathit{trieSequ}\ \mathit{te}\ \mathit{tz}\ \mathit{to})\ (\mathit{trieSequ}\ \mathit{te}'\ \mathit{tz}'\ \mathit{to}')} \\
 & = & \mathit{trieSequ}\ (\mathit{combine}\ c\ \mathit{te}\ \mathit{te}') \\
 & & \qquad (\mathit{mergeSequ}\ (\mathit{mergeFork}\ \mathit{mA})\ c\ \mathit{tz}\ \mathit{tz}') \\
 & & \qquad (\mathit{mA}\ (\mathit{mergeSequ}\ (\mathit{mergeFork}\ \mathit{mA})\ c)\ \mathit{to}\ \mathit{to}') \quad \square
\end{array}
$$

5.5.8 Properties

The functions on tries enjoy several properties which hold generically for all instances of K and which can be proved by fixed point induction.

$$
\begin{aligned}
lookup\langle K\rangle\ k\ (empty\langle K\rangle) &= nothing \\
lookup\langle K\rangle\ k\ (single\langle K\rangle\ (k_1, v_1)) &= \textbf{if}\ k == k_1\ \textbf{then}\ just\ v_1\ \textbf{else}\ nothing \\
lookup\langle K\rangle\ k\ (merge\langle K\rangle\ c\ t_1\ t_2) &= combine\ c\ (lookup\langle K\rangle\ k\ t_1)\ (lookup\langle K\rangle\ k\ t_2)
\end{aligned}
$$

The last law, for instance, states that looking up a key in the merge of two tries yields the same result as looking up the key in each trie separately and then combining the results. If the combining form c is associative,

$$c\ v_1\ (c\ v_2\ v_3) = c\ (c\ v_1\ v_2)\ v_3,$$

then $merge\langle K\rangle\ c$ is associative, as well. Furthermore, $empty\langle K\rangle$ is the left and the right unit of $merge\langle K\rangle\ c$:

$$
\begin{aligned}
merge\langle K\rangle\ c\ (empty\langle K\rangle)\ t &= t \\
merge\langle K\rangle\ c\ t\ (empty\langle K\rangle) &= t \\
merge\langle K\rangle\ c\ t_1\ (merge\langle K\rangle\ c\ t_2\ t_3) &= merge\langle K\rangle\ c\ (merge\langle K\rangle\ c\ t_1\ t_2)\ t_3.
\end{aligned}
$$

5.5.9 Related work

Knuth (1998) attributes the idea of a trie to Thue who introduced it in a paper about strings that do not contain adjacent repeated substrings (1912). De la Briandais recommended tries for computer searching (1959). The generalization of tries from strings to elements built according to an arbitrary signature was discovered by Wadsworth (1979) and others independently since. Connelly and Morris (1995) formalized the concept of a trie in a categorical setting: they showed that a trie is a functor and that the corresponding look-up function is a natural transformation.

The first implementation of generalized tries was given by Okasaki in his recent textbook on functional data structures (1998). Tries for parameterized types like lists or binary trees are represented as Standard ML functors. While this approach works for regular data types, it fails for nested data types such as *Sequ*. In the latter case data types of second-order kind are indispensable.

5.6 Generic memo tables

This section presents a generic implementation of memo functions that is based on a variation of digital search trees. A memo function can be seen as the composition of a tabulation function that creates a memo table and a look-up function that queries the table. We show that tabulation can be derived from look-up by inverse function construction. A memo table for a fixed argument type is a functor and look-up and tabulation are natural isomorphisms. We provide simple generic proofs of these properties. Contrary to the preceding section the implementation of memo table relies in a essential way on lazy evaluation.

5.6.1 Introduction

A *memo function* (Michie 1968) is like an ordinary function except that it caches previously computed values. If it is applied a second time to a particular argument, it immediately returns the cached result, rather than recomputing it. For storing arguments and results a memo function internally employs an index structure, the so-called *memo table*. In fact, a memo function can be seen as the composition of a *tabulation* function that creates a memo table and a *look-up* function that queries the table.

A memo table can be implemented in a variety of ways using, for instance, hashing or comparison-based search tree schemes. These approaches, however, have their drawbacks if the argument to a memo function is a compound value such as a list or a tree. Since comparing compound values is expensive, search tree schemes based on ordering are prohibitive. Hash tables are no viable alternative as hashing compound values is difficult. Furthermore, in case of collisions values must be checked for equality (though a hash-consing garbage collector (Appel and Goncalves 1993) may alleviate this problem). For memo functions with compound argument types *tries* are again the data structure of choice. Looking up a value in a trie takes time proportional to the size of the value. In particular, the running time is independent of the number of memoized values. In combination with lazy evaluation tries provide an elegant and efficient implementation of memo functions.

5.6.2 Signature

The signature of trie-based memo tables with associated look-up and tabulation functions is given by

$$\begin{array}{lcl} \mathit{Table}\langle K :: \star\rangle & :: & \star \to \star \\ \mathit{apply}\langle K\rangle & :: & \forall V\,.\,\mathit{Table}\langle K\rangle\ V \to (K \to V) \\ \mathit{tabulate}\langle K\rangle & :: & \forall V\,.\,(K \to V) \to \mathit{Table}\langle K\rangle\ V. \end{array}$$

The type $\mathit{Table}\langle K\rangle\ V$ represents memo tables that are indexed by values of type K and store values of type V. The function $\mathit{apply}\langle K\rangle$ is the associated look-up function: it takes a memo table and a key of type K and returns the associated value of type V. Its converse, $\mathit{tabulate}\langle K\rangle$, tabulates a given function with argument type K. Given this interface we can easily memoize a function of type $K \to V$:

$$\begin{array}{lcl} \mathit{memo}\langle K\rangle & :: & \forall V\,.\,(K \to V) \to (K \to V) \\ \mathit{memo}\langle K\rangle\, f & = & \mathit{apply}\langle K\rangle\ (\mathit{tabulate}\langle K\rangle\, f). \end{array}$$

The memoized version of f is simply $\mathit{memo}\langle K\rangle\ f$. It is worth noting that this technique depends in an essential way on *lazy evaluation*: if the type of keys is infinite, then $\mathit{tabulate}\langle K\rangle\ f$ produces a potentially infinite tree. We also require *full laziness* so that $\mathit{tabulate}\langle K\rangle\ f$ is evaluated only once even if it is queried several times. Haskell meets both requirements.

5.6.3 Memo tables

Memo tables are a simple variant of tries (for simplicity, we ignore the type constants *Char* and *Int*):

$$\begin{array}{lcl} \mathit{Table}\langle K :: \star\rangle & :: & \star \to \star \\ \mathit{Table}\langle 1\rangle & = & \Lambda V\,.\,V \\ \mathit{Table}\langle K_1 + K_2\rangle & = & \Lambda V\,.\,\mathit{Table}\langle K_1\rangle\ V \times \mathit{Table}\langle K_2\rangle\ V \\ \mathit{Table}\langle K_2 \times K_2\rangle & = & \Lambda V\,.\,\mathit{Table}\langle K_1\rangle\ (\mathit{Table}\langle K_2\rangle\ V). \end{array}$$

The type constructor $\mathit{Table}\langle K\rangle$ has kind $\star \to \star$. In fact, we will see in Section 5.6.6 that $\mathit{Table}\langle K\rangle$ satisfies the properties of a functor. In particular, the trie for the unit type is the identity functor, the trie for sums is a product of functors, and the trie for products is a composition of functors.

EXAMPLE 5.10 The memo table for the type of natural numbers is an *infinite list*.

$$\begin{aligned}
\mathit{Nat} &= \mathit{Fix}\ (\Lambda \mathit{Nat}\,.\,1 + \mathit{Nat}) \\
\mathit{TableNat} &= \mathit{Fix}\ (\Lambda \mathit{TableNat}\,.\,\Lambda V\,.\,V \times \mathit{TableNat}\ V)
\end{aligned}$$

In Haskell notation $\mathit{TableNat}$ reads

$$\mathbf{data}\ \mathit{TableNat}\ V \quad = \quad \mathit{nodeNat}\ V\ (\mathit{TableNat}\ V).$$

If we replace *nodeNat* by *cons* and add a case for *nil*, we obtain the familiar type of lists. Note that this instance, the use of infinite lists for memoizing functions on the natural numbers, already appears in Turner (1981). □

EXAMPLE 5.11 The memo table for binary numbers is an infinite binary tree

$$\begin{aligned}
\mathit{BNat} &= \mathit{Fix}\ (\Lambda \mathit{BNat}\,.\,1 + \mathit{BNat} + \mathit{BNat}) \\
\mathit{TableBNat} &= \mathit{Fix}\ (\Lambda \mathit{TableBNat}\,.\,\Lambda V\,.\,V \times \mathit{TableBNat}\ V \times \mathit{TableBNat}\ V)
\end{aligned}$$

and the corresponding Haskell type is given by

$$\mathbf{data}\ \mathit{TableBNat}\ V \quad = \quad \mathit{nodeBNat}\ V\ (\mathit{TableBNat}\ V)\ (\mathit{TableBNat}\ V) \quad \Box$$

EXAMPLE 5.12 Finally, let us consider a parameterized data type, the ubiquitous data type of lists. Since *List* is a type constructor, *TableList* is a 'higher-order' memo table that takes a trie for the base type A and yields a trie for $\mathit{List}\ A$.

$$\begin{aligned}
\mathit{List} &= \mathit{Fix}\ (\Lambda \mathit{List}\,.\,\Lambda A\,.\,1 + A \times \mathit{List}\ A) \\
\mathit{TableList} &= \mathit{Fix}\ (\Lambda \mathit{TableList}\,.\,\Lambda \mathit{TableA}\,.\,\Lambda V\,.\,V \times \mathit{TableA}\ (\mathit{TableList}\ \mathit{TableA}\ V))
\end{aligned}$$

Surprisingly, the type constructor *TableList* is isomorphic to the type of *generalized rose trees*. The corresponding Haskell type reads

$$\mathbf{data}\ \mathit{TableList}\ \mathit{TableA}\ V \quad = \quad \mathit{nodeList}\ V\ (\mathit{TableA}\ (\mathit{TableList}\ \mathit{TableA}\ V)) \quad \Box$$

5.6.4 Table look-up

The look-up function is given by the following generic definition.

$$\begin{aligned}
&\mathit{apply}\langle K\rangle && :: \forall V\,.\,\mathit{Table}\langle K\rangle\ V \to (K \to V) \\
&\mathit{apply}\langle 1\rangle\ t\ () && = t \\
&\mathit{apply}\langle K_1 + K_2\rangle\ (t_1, t_2)\ (\mathit{inl}\ k_1) && = \mathit{apply}\langle K_1\rangle\ t_1\ k_1 \\
&\mathit{apply}\langle K_1 + K_2\rangle\ (t_1, t_2)\ (\mathit{inr}\ k_2) && = \mathit{apply}\langle K_2\rangle\ t_2\ k_2 \\
&\mathit{apply}\langle K_1 \times K_2\rangle\ t\ (k_1, k_2) && = \mathit{apply}\langle K_2\rangle\ (\mathit{apply}\langle K_1\rangle\ t\ k_1)\ k_2
\end{aligned}$$

Note that *apply* is essentially the function *lookup* of Section 5.5.6 with the two arguments reversed:

$$\begin{aligned}
&\mathit{lookup}\langle K\rangle && :: \forall V\,.\,K \to \mathit{Table}\langle K\rangle\ V \to V \\
&\mathit{lookup}\langle 1\rangle\ () && = \mathit{id} \\
&\mathit{lookup}\langle K_1 + K_2\rangle\ (\mathit{inl}\ k_1) && = \mathit{lookup}\langle K_1\rangle\ k_1 \cdot \mathit{outl} \\
&\mathit{lookup}\langle K_1 + K_2\rangle\ (\mathit{inr}\ k_2) && = \mathit{lookup}\langle K_2\rangle\ k_2 \cdot \mathit{outr} \\
&\mathit{lookup}\langle K_1 \times K_2\rangle\ (k_1, k_2) && = \mathit{lookup}\langle K_2\rangle\ k_2 \cdot \mathit{lookup}\langle K_1\rangle\ k_1\,.
\end{aligned}$$

Thus, on the unit type the look-up function is the identity, on sums it selects the appropriate memo table, and on products it composes the look-up functions for the components.

Example 5.13 Querying a memo table for the natural numbers works as follows.

$$\begin{array}{lcl} \mathit{applyNat} & :: & \forall V \,.\, \mathit{TableNat}\ V \to (\mathit{Nat} \to V) \\ \mathit{applyNat}\ (\mathit{nodeNat}\ \mathit{tz}\ \mathit{ts})\ \mathit{zero} & = & \mathit{tz} \\ \mathit{applyNat}\ (\mathit{nodeNat}\ \mathit{tz}\ \mathit{ts})\ (\mathit{succ}\ n) & = & \mathit{applyNat}\ \mathit{ts}\ n \end{array}$$

Recall that elements of *TableNat* are infinite lists. Consequently, *applyNat* corresponds to list indexing, written (!!) in Haskell. □

Example 5.14 The look-up function for binary numbers corresponds to tree indexing (a binary number is interpreted as a path into a binary tree).

$$\begin{array}{lcl} \mathit{applyBin} & :: & \forall V \,.\, \mathit{TableBNat}\ V \to (\mathit{BNat} \to V) \\ \mathit{applyBin}\ (\mathit{nodeBNat}\ \mathit{tn}\ \mathit{to}\ \mathit{tt})\ \mathit{endB} & = & \mathit{tn} \\ \mathit{applyBin}\ (\mathit{nodeBNat}\ \mathit{tn}\ \mathit{to}\ \mathit{tt})\ (\mathit{zeroB}\ b) & = & \mathit{applyBin}\ \mathit{to}\ b \\ \mathit{applyBin}\ (\mathit{nodeBNat}\ \mathit{tn}\ \mathit{to}\ \mathit{tt})\ (\mathit{oneB}\ b) & = & \mathit{applyBin}\ \mathit{tt}\ b \quad \Box \end{array}$$

Example 5.15 As the final example, consider the look-up function for lists.

$$\begin{array}{lcl} \mathit{applyList} & :: & \forall \mathit{TA}\ A \,.\, (\forall V \,.\, \mathit{TA}\ V \to (A \to V)) \\ & & \qquad \to (\forall W \,.\, \mathit{TableList}\ \mathit{TA}\ W \to (\mathit{List}\ A \to W)) \\ \mathit{applyList}\ \mathit{applyA}\ (\mathit{nodeList}\ \mathit{tn}\ \mathit{tc})\ \mathit{nil} & = & \mathit{tn} \\ \mathit{applyList}\ \mathit{applyA}\ (\mathit{nodeList}\ \mathit{tn}\ \mathit{tc})\ (\mathit{cons}\ a\ \mathit{as}) & & \\ & = & \mathit{applyList}\ \mathit{applyA}\ (\mathit{applyA}\ \mathit{tc}\ a)\ \mathit{as} \end{array}$$

Since *List* is a parametric type, *applyList* is a 'higher-order' look-up function that takes a look-up function for the base type A and yields a lookup function for $\mathit{List}\ A$. □

5.6.5 Tabulation

Tabulation is the inverse of look-up and, in fact, we can derive its definition by inverse function construction. For the derivation we use a slight reformulation of *apply* that allows for more structured calculations.

$$\begin{array}{lcl} \mathit{apply}\langle K\rangle & :: & \forall V \,.\, \mathit{Table}\langle K\rangle\ V \to (K \to V) \\ \mathit{apply}\langle 1\rangle\ t & = & \lambda() \,.\, t \\ \mathit{apply}\langle K_1 + K_2\rangle\ t & = & \mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ t) \mathbin{\nabla} \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ t) \\ \mathit{apply}\langle K_1 \times K_2\rangle\ t & = & \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ t) \end{array}$$

We specify *tabulate* as the right inverse of *apply*:

$$\mathit{apply}\langle K\rangle\ (\mathit{tabulate}\langle K\rangle\ f) \quad = \quad f\,.$$

As usual, we proceed by case analysis on K.

- **Case** $K = 1$:

$$\begin{array}{ll} & \mathit{apply}\langle 1\rangle\ (\mathit{tabulate}\langle 1\rangle\ f) = f \\ \equiv & \quad \{\ \text{definition of } \mathit{apply}\ \} \\ & \lambda() \,.\, \mathit{tabulate}\langle 1\rangle\ f = f \\ \equiv & \quad \{\ \text{extensionality: } f_1 = f_2 :: 1 \to A \ \equiv\ f_1\ () = f_2\ () :: A\ \} \\ & \mathit{tabulate}\langle 1\rangle\ f = f\ (). \end{array}$$

- **Case** $K = K_1 + K_2$: let $t = \mathit{tabulate}\langle K_1 + K_2\rangle\ f$, then

$$\begin{array}{cl}
 & \mathit{apply}\langle K_1 + K_2\rangle\ t = f \\
\equiv & \quad \{\ \text{definition of}\ \mathit{apply}\ \} \\
 & \mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ t) \triangledown \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ t) = f \\
\equiv & \quad \{\ \text{universal property of coproducts}\ \} \\
 & \mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ t) = f \cdot \mathit{inl} \wedge \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ t) = f \cdot \mathit{inr} \\
\subset & \quad \{\ \text{specification}\ \} \\
 & \mathit{outl}\ t = \mathit{tabulate}\langle K_1\rangle\ (f \cdot \mathit{inl}) \wedge \mathit{outr}\ t = \mathit{tabulate}\langle K_2\rangle\ (f \cdot \mathit{inr}) \\
\equiv & \quad \{\ \text{surjective pairing:}\ z = (x_1, x_2) \equiv \mathit{outl}\ z = x_1 \wedge \mathit{outr}\ z = x_2\ \} \\
 & t = (\mathit{tabulate}\langle K_1\rangle\ (f \cdot \mathit{inl}), \mathit{tabulate}\langle K_2\rangle\ (f \cdot \mathit{inr})).
\end{array}$$

  Note that we use both the universal property of coproducts and the universal property of products (of which surjective pairing is a special case).

- **Case** $K = K_1 \times K_2$: let $t = \mathit{tabulate}\langle K_1 \times K_2\rangle\ f$, then

$$\begin{array}{cl}
 & \mathit{apply}\langle K_1 \times K_2\rangle\ t = f \\
\equiv & \quad \{\ \text{definition of}\ \mathit{apply}\ \} \\
 & \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ t) = f \\
\subset & \quad \{\ \text{exponentials:}\ \mathit{uncurry} \cdot \mathit{curry} = \mathit{id}\ \} \\
 & \mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ t = \mathit{curry}\ f \\
\subset & \quad \{\ \text{specification}\ \} \\
 & \mathit{apply}\langle K_1\rangle\ t = \mathit{tabulate}\langle K_2\rangle \cdot \mathit{curry}\ f \\
\subset & \quad \{\ \text{specification}\ \} \\
 & t = \mathit{tabulate}\langle K_1\rangle\ (\mathit{tabulate}\langle K_2\rangle \cdot \mathit{curry}\ f).
\end{array}$$

To summarize, we have calculated the following definition of *tabulate*.

$$\begin{array}{lcl}
\mathit{tabulate}\langle K\rangle & :: & \forall V\,.\,(K \to V) \to \mathit{Table}\langle K\rangle\ V \\
\mathit{tabulate}\langle 1\rangle\ f & = & f\ () \\
\mathit{tabulate}\langle K_1 + K_2\rangle\ f & = & (\mathit{tabulate}\langle K_1\rangle\ (f \cdot \mathit{inl}), \mathit{tabulate}\langle K_2\rangle\ (f \cdot \mathit{inr})) \\
\mathit{tabulate}\langle K_1 \times K_2\rangle\ f & = & \mathit{tabulate}\langle K_1\rangle\ (\mathit{tabulate}\langle K_2\rangle \cdot \mathit{curry}\ f)
\end{array}$$

The last equation becomes more readable if we convert it into a pointwise style.

$$\mathit{tabulate}\langle K_1 \times K_2\rangle\ f \quad = \quad \mathit{tabulate}\langle K_1\rangle\ (\lambda k_1\,.\,\mathit{tabulate}\langle K_2\rangle\ (\lambda k_2\,.\,f\ (k_1, k_2)))$$

Two points are in order.

First, the second calculation makes essential use of the universal property of coproducts. Alas, Haskell's natural semantic model, the category $\mathcal{C}po$ of pointed, complete partial orders and continuous functions, has no categorical coproduct. In other words, in Haskell $\mathit{apply}\langle K\rangle\ (\mathit{tabulate}\langle K\rangle\ f) = f$ is only valid for so-called *hyper-strict* functions that completely evaluate their arguments. In the context of a lazy language this need for hyper-strictness is somewhat ironic. The intuition is that *all* information about the result of a memoized function is in the leaves of the corresponding trie.

Note that an appropriate theoretical setting for the calculations is the category $\mathcal{C}po_\bot$ of pointed, complete partial orders and *strict* continuous functions, which has categorical products (the cartesian product '$\times$'), categorical coproducts (the coalesced sum '$\oplus$') and is monoidally closed (the smash product '$\otimes$' and the space '$\circ\!\!\rightarrow$' of strict continuous functions form a monoidal closure). Thus, memo tables are actually based on the following isomorphisms:

$$\begin{array}{lcl} 1_\perp \circ\!\!\rightarrow V & \cong & V \\ (K_1 \oplus K_2) \circ\!\!\rightarrow V & \cong & (K_1 \circ\!\!\rightarrow V) \times (K_2 \circ\!\!\rightarrow V) \\ (K_1 \otimes K_2) \circ\!\!\rightarrow V & \cong & K_1 \circ\!\!\rightarrow (K_2 \circ\!\!\rightarrow V), \end{array}$$

where $1_\perp = \{\perp, ()\}$. The isomorphisms make precise that memoization operates on strict functions but its implementation requires lazy evaluation: a trie for a 'strict' sum is a 'lazy' pair of tries. We could maintain this distinction in Haskell using strictness annotations (TNat is really the memo table for the flat domain $\mathbb{N}_\perp$ given by **data** $\mathit{Nat} = \mathit{zero} \mid \mathit{succ}\,!\,\mathit{Nat}$) but we refrain from being that pedantic.

Second, the calculations show that tabulation is the right inverse of look-up. The converse can be shown using a straightforward fixed point induction. That said, we notice that the case $K = 0$, where $0 = \{\perp\}$ is the 'bottom' type, is missing in the derivation above. Fortunately, $\mathit{apply}\langle 0\rangle\ (\mathit{tabulate}\langle 0\rangle\ f) = f$ holds trivially since '0' is the initial object in $\mathcal{C}po_\perp$, that is, for each type V there is a unique *strict* function of type $0 \to V$.

Example 5.16 The tabulation function for natural numbers is a one-liner.

$$\begin{array}{lcl} \mathit{tabulateNat} & :: & \forall V\,.\,(\mathit{Nat} \to V) \to \mathit{TableNat}\ V \\ \mathit{tabulateNat}\ f & = & \mathit{nodeNat}\ (f\ \mathit{zero})\ (\mathit{tabulateNat}\ (f \cdot \mathit{succ})) \end{array}$$

The standard toy example of memoization is the Fibonacci function.

$$\begin{array}{lcl} \mathit{fib} & :: & \mathit{Nat} \to \mathit{Nat} \\ \mathit{fib}\ \mathit{zero} & = & \mathit{zero} \\ \mathit{fib}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{succ}\ \mathit{zero} \\ \mathit{fib}\ (\mathit{succ}\ (\mathit{succ}\ n)) & = & \mathit{fib}\ n + \mathit{fib}\ (\mathit{succ}\ n) \end{array}$$

Its time complexity can be improved from exponential to quadratic if the recursive calls are replaced by table lookups.

$$\begin{array}{lcl} \mathit{fib} & :: & \mathit{Nat} \to \mathit{Nat} \\ \mathit{fib}\ \mathit{zero} & = & \mathit{zero} \\ \mathit{fib}\ (\mathit{succ}\ \mathit{zero}) & = & \mathit{succ}\ \mathit{zero} \\ \mathit{fib}\ (\mathit{succ}\ (\mathit{succ}\ n)) & = & \mathit{memo\text{-}fib}\ n + \mathit{memo\text{-}fib}\ (\mathit{succ}\ n) \\ \mathit{memo\text{-}fib} & :: & \mathit{Nat} \to \mathit{Nat} \\ \mathit{memo\text{-}fib} & = & \mathit{applyNat}\ (\mathit{tabulateNat}\ \mathit{fib}) \quad \square \end{array}$$

Example 5.17 Tabulating a function of type $\mathit{Bin} \to V$ is equally easy.

$$\begin{array}{lcl} \mathit{tabulateBin} & :: & \forall V\,.\,(\mathit{BNat} \to V) \to \mathit{TableBNat}\ V \\ \mathit{tabulateBin}\ f & = & \mathit{nodeBNat}\ (f\ \mathit{endB})\ (\mathit{tabulateBin}\ (f \cdot \mathit{zeroB}))\ (\mathit{tabulateBin}\ (f \cdot \mathit{oneB})) \quad \square \end{array}$$

Example 5.18 Finally, for parametric lists we obtain a 'higher-order' tabulation function.

$$\begin{array}{lcl} \mathit{tabulateList} & :: & \forall \mathit{TA}\ A\,.\,(\forall V\,.\,(A \to V) \to \mathit{TA}\ V) \\ & & \qquad \to (\forall W\,.\,(\mathit{List}\ A \to W) \to \mathit{TableList}\ \mathit{TA}\ W) \\ \mathit{tabulateList}\ \mathit{tabulateA}\ f & = & \mathit{nodeList}\ (f\ \mathit{nil})\ (\mathit{tabulateA}\ (\lambda a \to \\ & & \qquad \mathit{tabulateList}\ \mathit{tabulateA}\ (\lambda as \to f\ (\mathit{cons}\ a\ as)))) \end{array}$$

Using *TableList* we can memoize functions that operate on lists. The following dynamic programming problem, *optimal matrix multiplication*, may serve as an example. Given a sequence of matrix dimensions $[d_0, \ldots, d_n]$, the problem is to find the least cost for multiplying out a sequence of matrices $M_1 * \cdots * M_n$ where the dimension of M_i is $d_{i-1} \times d_i$. We assume that multiplying an $i \times j$ matrix by an $j \times k$ matrix costs $i \times j \times k$. The following Haskell program implements a straightforward, but exponential solution.

$$
\begin{array}{lcl}
\mathit{cost} & :: & \mathit{List\ Nat} \to \mathit{Nat} \\
\mathit{cost}\ d & & \\
\quad \mid n \leqslant 1 & = & 0 \\
\quad \mid \mathit{otherwise} & = & \mathit{minimum}\,[\,\mathit{cost}\ (\mathit{take}\ (k+1)\ d) \\
& & \qquad\qquad +d\ !!\ 0 \times d\ !!\ k \times d\ !!\ n \\
& & \qquad\qquad +\mathit{cost}\ (\mathit{drop}\ k\ d) \mid k \leftarrow [1 \mathinner{..} n-1]] \\
\quad \textbf{where}\ n & = & \mathit{length}\ d - 1
\end{array}
$$

Memoizing the recursive calls improves the complexity from exponential to polynomial in the size of the input.

$$
\begin{array}{lcl}
\mathit{memo\text{-}cost} & :: & \mathit{List\ Nat} \to \mathit{Nat} \\
\mathit{memo\text{-}cost} & = & (\mathit{applyList\ applyNat})\ ((\mathit{tabulateList\ tabulateNat})\ \mathit{cost}) \\
\mathit{cost} & :: & \mathit{List\ Nat} \to \mathit{Nat} \\
\mathit{cost}\ d & & \\
\quad \mid n \leqslant 1 & = & 0 \\
\quad \mid \mathit{otherwise} & = & \mathit{minimum}\,[\,\mathit{memo\text{-}cost}\ (\mathit{take}\ (k+1)\ d) \\
& & \qquad\qquad +d\ !!\ 0 \times d\ !!\ k \times d\ !!\ n \\
& & \qquad\qquad +\mathit{memo\text{-}cost}\ (\mathit{drop}\ k\ d) \mid k \leftarrow [1 \mathinner{..} n-1]] \\
\quad \textbf{where}\ n & = & \mathit{length}\ d - 1
\end{array}
$$

An ad-hoc variant of this code appears in O'Donnell (1985). □

EXAMPLE 5.19 The function *memo-cost* defined in the previous example maintains a global memo table. This comes at a considerable cost: recall that functions on the natural numbers are memoized using infinite lists and note that the matrix dimensions $d_0, \ldots, d_n$ index these lists. A more efficient alternative both in time and in space is to maintain a local memo table.

$$
\begin{array}{lcl}
\mathit{cost} & :: & \mathit{List\ Int} \to \mathit{Int} \\
\mathit{cost}\ d & = & \mathit{memo\text{-}c}\ (0, n) \\
\quad \textbf{where} & & \\
\quad n & = & \mathit{length}\ d - 1 \\
\quad c & :: & (\mathit{Nat}, \mathit{Nat}) \to \mathit{Int} \\
\quad c\ (i, j) & & \\
\qquad \mid i + 1 \geqslant j & = & 0 \\
\qquad \mid \mathit{otherwise} & = & \mathit{minimum}\,[\,\mathit{memo\text{-}c}\ (i, k) \\
& & \qquad\qquad + d\ !!\ i \times d\ !!\ k \times d\ !!\ j \\
& & \qquad\qquad + \mathit{memo\text{-}c}\ (k, j) \mid k \leftarrow [i+1 \mathinner{..} j-1]] \\
\quad \mathit{memo\text{-}c} & :: & (\mathit{Nat}, \mathit{Nat}) \to \mathit{Int} \\
\quad \mathit{memo\text{-}c}\ (i, j) & = & \mathit{applyNat}\ (\mathit{applyNat}\ (\\
& & \quad \mathit{tabulateNat}\ (\lambda i' \to \mathit{tabulateNat}\ (\lambda j' \to c\ (i', j'))))\ i)\ j
\end{array}
$$

Since the sequence of matrix dimensions d is fixed in the body of *cost*, sublists of d can be represented by pairs of list indices. Consequently, a much smaller memo table suffices: *memo-c* uses a table of type *TableNat* (*TableNat Int*) that is indexed by pairs of list indices (which are small) rather than by sequences of matrix dimensions (which may be be very large). The resulting code corresponds closely to the standard dynamic programming solution, see, for instance, Rabhi and Lapalme (1999). □

5.6.6 Properties

For a fixed K, the type constructor $\mathit{Table}\langle K\rangle$ satisfies the properties of a functor (it is an endo functor of $\mathcal{C}po_{\perp}$). Its functorial action on arrows is given by

$$\begin{array}{lcl} \mathit{table}\langle K\rangle & :: & \forall V\ W.\,(V \to W) \to (\mathit{Table}\langle K\rangle\ V \to \mathit{Table}\langle K\rangle\ W) \\ \mathit{table}\langle 1\rangle\, f & = & f \\ \mathit{table}\langle K_1 + K_2\rangle\, f & = & \mathit{table}\langle K_1\rangle\, f \times \mathit{table}\langle K_2\rangle\, f \\ \mathit{table}\langle K_1 \times K_2\rangle\, f & = & \mathit{table}\langle K_1\rangle\ (\mathit{table}\langle K_2\rangle\, f). \end{array}$$

The functor laws

$$\begin{array}{lcl} \mathit{table}\langle K\rangle\ \mathit{id} & = & \mathit{id} \\ \mathit{table}\langle K\rangle\ (f \cdot g) & = & \mathit{table}\langle K\rangle\, f \cdot \mathit{table}\langle K\rangle\ g \end{array}$$

can be shown using straightforward fixed point inductions.

The functions $\mathit{apply}\langle K\rangle$ and $\mathit{tabulate}\langle K\rangle$ are then natural isomorphisms between $(-)^K$ and $\mathit{Table}\langle K\rangle$. Note that the functorial action of $(-)^K$ is postcomposition given by $\mathit{post}\, f = \mathit{curry}\ (f \cdot \mathit{eval})$ where eval is function application. The naturality conditions are

$$\begin{array}{lcl} \mathit{apply}\langle K\rangle \cdot \mathit{table}\langle K\rangle\, f & = & \mathit{post}\, f \cdot \mathit{apply}\langle K\rangle \\ \mathit{tabulate}\langle K\rangle \cdot \mathit{post}\, f & = & \mathit{table}\langle K\rangle\, f \cdot \mathit{tabulate}\langle K\rangle. \end{array}$$

The proofs below are based on the following pointwise variants.

$$\begin{array}{lcl} \mathit{apply}\langle K\rangle\ (\mathit{table}\langle K\rangle\ f\ t) & = & f \cdot \mathit{apply}\langle K\rangle\ t \\ \mathit{tabulate}\langle K\rangle\ (f \cdot g) & = & \mathit{table}\langle K\rangle\, f\ (\mathit{tabulate}\langle K\rangle\ g) \end{array}$$

An immediate consequence of the second naturality property is, for instance,

$$\mathit{tabulate}\langle K\rangle\, f \quad = \quad \mathit{table}\langle K\rangle\, f\ (\mathit{tabulate}\langle K\rangle\ \mathit{id}).$$

Thus, instead of tabulating f we can tabulate id and then map f on the resulting memo table. Since some types allow for a more efficient implementation of $\mathit{tabulate}\langle K\rangle\ \mathit{id}$, applying the law from left to right may be an optimization. We prove $\mathit{apply}\langle K\rangle\ (\mathit{table}\langle K\rangle\ f\ t) = f \cdot \mathit{apply}\langle K\rangle\ t$ by fixed point induction on K. The second naturality property then follows immediately since $\mathit{apply}\langle K\rangle$ and $\mathit{tabulate}\langle K\rangle$ are mutually inverse.

- **Case** $K = 0$: the proposition holds trivially for strict f since generic functions are strict in their type arguments.
- **Case** $K = 1$:

$$\begin{array}{ll} & \mathit{apply}\langle 1\rangle\ (\mathit{table}\langle 1\rangle\ f\ t) \\ = & \quad \{\ \text{definition of } \mathit{apply}\ \} \end{array}$$

$$
\begin{aligned}
& \lambda() \,.\, \mathit{table}\langle 1\rangle\ f\ t \\
= \quad & \{\ \text{definition of } \mathit{table}\ \} \\
& \lambda() \,.\, f\ t \\
= \quad & \{\ \text{extensionality: } g_1 = g_2 :: 1 \to A \ \equiv\ g_1\ () = g_2\ () :: A\ \} \\
& f \cdot (\lambda() \,.\, t) \\
= \quad & \{\ \text{definition of } \mathit{apply}\ \} \\
& f \cdot \mathit{apply}\langle 1\rangle\ t.
\end{aligned}
$$

- **Case** $K = K_1 + K_2$:

$$
\begin{aligned}
& \mathit{apply}\langle K_1 + K_2\rangle\ (\mathit{table}\langle K_1 + K_2\rangle\ f\ t) \\
= \quad & \{\ \text{definition of } \mathit{apply}\ \} \\
& \mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ (\mathit{table}\langle K_1 + K_2\rangle\ f\ t)) \mathbin{\triangledown} \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ (\mathit{table}\langle K_1 + K_2\rangle\ f\ t)) \\
= \quad & \{\ \text{definition of } \mathit{table} \text{ and } \times\text{-computation laws}\ \} \\
& \mathit{apply}\langle K_1\rangle\ (\mathit{table}\langle K_1\rangle\ f\ (\mathit{outl}\ t)) \mathbin{\triangledown} \mathit{apply}\langle K_2\rangle\ (\mathit{table}\langle K_2\rangle\ f\ (\mathit{outr}\ t)) \\
= \quad & \{\ \text{ex hypothesi}\ \} \\
& (f \cdot \mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ t)) \mathbin{\triangledown} (f \cdot \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ t)) \\
= \quad & \{\ \triangledown\text{-fusion law}\ \} \\
& f \cdot (\mathit{apply}\langle K_1\rangle\ (\mathit{outl}\ t) \mathbin{\triangledown} \mathit{apply}\langle K_2\rangle\ (\mathit{outr}\ t)) \\
= \quad & \{\ \text{definition of } \mathit{apply}\ \} \\
& f \cdot \mathit{apply}\langle K_1 + K_2\rangle\ t.
\end{aligned}
$$

- **Case** $K = K_1 \times K_2$:

$$
\begin{aligned}
& \mathit{apply}\langle K_1 \times K_2\rangle\ (\mathit{table}\langle K_1 \times K_2\rangle\ f\ t) \\
= \quad & \{\ \text{definition of } \mathit{apply}\ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ (\mathit{table}\langle K_1 \times K_2\rangle\ f\ t)) \\
= \quad & \{\ \text{definition of } \mathit{table}\ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ (\mathit{table}\langle K_1\rangle\ (\mathit{table}\langle K_2\rangle\ f)\ t)) \\
= \quad & \{\ \text{ex hypothesi}\ \} \\
& \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{table}\langle K_2\rangle\ f \cdot \mathit{apply}\langle K_1\rangle\ t) \\
= \quad & \{\ \text{ex hypothesi}\ \} \\
& \mathit{uncurry}\ (\mathit{post}\ f \cdot \mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ t) \\
= \quad & \{\ \text{proof obligation, see below}\ \} \\
& f \cdot \mathit{uncurry}\ (\mathit{apply}\langle K_2\rangle \cdot \mathit{apply}\langle K_1\rangle\ t) \\
= \quad & \{\ \text{definition of } \mathit{apply}\ \} \\
& f \cdot \mathit{apply}\langle K_1 \times K_2\rangle\ t.
\end{aligned}
$$

It remains to show $f \cdot \mathit{uncurry}\ g = \mathit{uncurry}\ (\mathit{post}\ f \cdot g)$, which is equivalent to $\mathit{curry}\ (f \cdot \mathit{uncurry}\ g) = \mathit{post}\ f \cdot g$.

$$
\begin{aligned}
& \mathit{curry}\ (f \cdot \mathit{uncurry}\ g) \\
= \quad & \{\ \text{definition of } \mathit{uncurry}\ \} \\
& \mathit{curry}\ (f \cdot \mathit{eval} \cdot (g \times \mathit{id}))
\end{aligned}
$$

$$
\begin{aligned}
= &\quad \{ \text{ curry fusion law: } \mathit{curry}\ h \cdot k = \mathit{curry}\ (h \cdot (k \times \mathit{id})) \ \} \\
&\mathit{curry}\ (f \cdot \mathit{eval}) \cdot g \\
= &\quad \{ \text{ definition of } \mathit{post} \ \} \\
&\mathit{post}\ f \cdot g
\end{aligned}
$$

Chapter 6

Generic Haskell

This chapter is concerned with the details and pragmatics of adding generic programming to Haskell. Interestingly, Haskell already provides a rudimentary form of genericity in form of the **deriving** mechanism—for a discussion of this feature and of Haskell's class system in general, see Section 2.2. By attaching a **deriving** clause to a data type declaration instance declarations are generated automatically by the compiler. Unfortunately, this feature is rather ad-hoc: the derived code is specified only informally in an appendix of the language definition (Peyton Jones and Hughes 1999) and more severely the **deriving** mechanism is restricted to a fixed set of built-in classes. Both problems can be overcome using generic definitions for default method declarations. An extension of Haskell along these lines is described in (Hinze and Peyton Jones 2000). In the sequel we discuss a less tight integration: we show how to translate instances of generic definitions into ordinary Haskell definitions. Overall, we are more concerned with implementation techniques and less with language design issues.

This chapter is organized as follows. Section 6.1 discusses the specialization of generic values using Haskell as a target language. Thereby we restrict ourselves to MPC-style definitions as they are more general than POPL-style definitions (nonetheless, we will use POPL-style definitions for the examples). Section 6.2 introduces two extensions to generic definitions that are useful or even necessary in a concrete implementation: ad-hoc definitions to cope with abstract data types and provisions for accessing constructor names and record labels.

6.1 Implementation

The polymorphic λ-calculus is the language of choice for the theoretical treatment of generic definitions as it offers rank-n polymorphism, which is required for specializing higher-order kinded data types. We additionally equipped it with a liberal notion of type equivalence so that we can interpret the type definition $\mathit{List}\ A = 1 + A \times \mathit{List}\ A$ as an equality rather than as an isomorphism.

Haskell—or rather, extensions of Haskell come quite close to this ideal language. The Glasgow Haskell Compiler, GHC, (Team 2000), the Haskell B. Compiler, HBC, (Augustsson 1998) and the Haskell interpreter Hugs (Jones and Peterson 1999) provide rank-2 type signatures and local universal quantification in data types. We will see in Section 6.1.4 that the latter feature can be used to encode rank-n types. There is, however, one fundamental difference between Haskell and (our presentation) of the polymorphic λ-calculus: Haskell's notion of type equivalence is based on *name equivalence* while the polymorphic λ-calculus employs *structural equivalence*. Sections 6.1.1–6.1.3 explain how to adapt the techniques of Chapter 3 to type systems that are based on name equivalence.

6.1.1 Generic representation types

Consider the Haskell data type of parametric lists:

$$\textbf{data}\ \mathit{List}\ A \quad = \quad \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A).$$

We have modelled this declaration (see Section 2.5.1) by the type term

$$\mathit{Fix}\ (\Lambda \mathit{List}\,.\,\Lambda A\,.\,1 + A \times \mathit{List}\ A).$$

Since the equivalence of type terms is based on structural equivalence, captured by the relation '$\approx$', we have, in particular, that $\mathit{List}\ A \approx 1 + A \times \mathit{List}\ A$. It is important to note that the specialization of generic values described in Section 3.1.3 makes essential use of this fact: the *List* instance of *poly* given by (omitting type abstractions and type applications)

$$\mathit{fix}\ (\lambda \mathit{poly}_{\mathit{List}}\,.\,\lambda \mathit{poly}_{A}\,.\,\mathit{poly}_{+}\ \mathit{poly}_{1}\ (\mathit{poly}_{\times}\ \mathit{poly}_{A}\ (\mathit{poly}_{\mathit{List}}\ \mathit{poly}_{A})))$$

only works under the proviso that $\mathit{List}\ A \approx 1 + A \times \mathit{List}\ A$. Alas, in Haskell $\mathit{List}\ A$ is not equivalent to $1 + A \times \mathit{List}\ A$ as each **data** declaration introduces a new distinct type. Even the type *Liste* defined by

$$\textbf{data}\ \mathit{Liste}\ A \quad = \quad \mathit{Vide} \mid \mathit{Constructeur}\ A\ (\mathit{Liste}\ A)$$

is not equivalent to *List*. Furthermore, Haskell's **data** construct works with n-ary sums and products whereas generic definitions operate on binary sums and products. The bottom line of all this is that when generating instances we additionally have to introduce conversion functions which perform the impedance-matching. This and the next two sections explain how to do this in a systematic way.

To begin with we introduce so-called *generic representation types*, which mediate between the two representations. For instance, the generic representation type for *List*, which we will call List°, is given by

$$\textbf{type}\ \mathit{List}^{\circ}\ A \quad = \quad 1 + A \times \mathit{List}\ A.$$

As to be expected our generic representation type constructors are just unit, sum and product. In particular, there is no recursion operator. Thus, we observe that List° is a non-recursive type synonym: *List* (not List°) appears on the right-hand side. So List° is not a recursive type; rather, it expresses the 'top layer' of a list structure, leaving the original *List* to do the rest.

The type constructor List° is (more or less) isomorphic to *List*. To make the isomorphism explicit, let us write functions that convert to and fro:

$$\begin{array}{lcl}
\mathit{from}_{\mathit{List}} & :: & \forall A\,.\,\mathit{List}\ A \to \mathit{List}^{\circ}\ A \\
\mathit{from}_{\mathit{List}}\ \mathit{nil} & = & \mathit{inl}\ () \\
\mathit{from}_{\mathit{List}}\ (\mathit{cons}\ x\ \mathit{xs}) & = & \mathit{inr}\ (x, \mathit{xs}) \\
\mathit{to}_{\mathit{List}} & :: & \forall A\,.\,\mathit{List}^{\circ}\ A \to \mathit{List}\ A \\
\mathit{to}_{\mathit{List}}\ (\mathit{inl}\ ()) & = & \mathit{nil} \\
\mathit{to}_{\mathit{List}}\ (\mathit{inr}\ (x, \mathit{xs})) & = & \mathit{cons}\ x\ \mathit{xs}.
\end{array}$$

Though these are non-generic functions, it is not hard to generate them mechanically. That is what we turn our attention to now.

Since the generic definitions work with *binary* sums and products, algebraic data types with many constructors, each of which has many fields, must be encoded as nested uses of sum and product. There are many possible encodings. For concreteness, we use a simple *linear encoding*: for

$$\mathbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}$$

we generate:

$$\mathbf{type}\ B^\circ\ A_1\ \ldots\ A_m \quad = \quad \Sigma\ (\Pi\ T_{11}\ \ldots\ T_{1m_1})\ \cdots\ (\Pi\ T_{n1}\ \ldots\ T_{nm_n})$$

where 'Σ' and 'Π' are defined

$$\begin{aligned}
\Sigma\ T_1\ \ldots\ T_n &= \begin{cases} T_1 & \text{if } n = 1 \\ T_1 + \Sigma\ T_2\ \ldots\ T_n & \text{if } n > 1 \end{cases} \\
\Pi\ T_1\ \ldots\ T_n &= \begin{cases} 1 & \text{if } n = 0 \\ T_1 & \text{if } n = 1 \\ T_1 \times \Pi\ T_2\ \ldots\ T_n & \text{if } n > 1. \end{cases}
\end{aligned}$$

Note that this encoding corresponds closely to the scheme introduced in Section 2.5.1 except that here the argument types of the constructors are *not* recursively encoded. The conversion functions from_B and to_B are then given by

$$\begin{aligned}
&\mathit{from}_B && :: \forall A_1\ \ldots\ A_m\,.\,B\ A_1\ \ldots\ A_m \to B^\circ\ A_1\ \ldots\ A_m \\
&\mathit{from}_B\ (k_1\ x_{11}\ \ldots\ x_{1m_1}) && = \mathit{in}_1^n\ (\mathit{tuple}\ x_{11}\ \ldots\ x_{1m_1}) \\
&\ldots && \\
&\mathit{from}_B\ (k_n\ x_{n1}\ \ldots\ x_{nm_n}) && = \mathit{in}_n^n\ (\mathit{tuple}\ x_{n1}\ \ldots\ x_{nm_n}) \\
&\mathit{to}_B && :: \forall A_1\ \ldots\ A_m\,.\,B^\circ\ A_1\ \ldots\ A_m \to B\ A_1\ \ldots\ A_m \\
&\mathit{to}_B\ (\mathit{in}_1^n\ (\mathit{tuple}\ x_{11}\ \ldots\ x_{1m_1})) && = k_1\ x_{11}\ \ldots\ x_{1m_1} \\
&\ldots && \\
&\mathit{to}_B\ (\mathit{in}_n^n\ (\mathit{tuple}\ x_{n1}\ \ldots\ x_{nm_n})) && = k_n\ x_{n1}\ \ldots\ x_{nm_n}
\end{aligned}$$

where

$$\begin{aligned}
\mathit{in}_i^n\ t &= \begin{cases} t & \text{if } n = 1 \\ \mathit{inl}\ t & \text{if } n > 1 \wedge i = 1 \\ \mathit{inr}\ (\mathit{in}_{i-1}^{n-1}\ t) & \text{if } n > 1 \wedge i > 1 \end{cases} \\
\mathit{tuple}\ t_1\ \ldots\ t_n &= \begin{cases} () & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ (t_1, \mathit{tuple}\ t_2\ \ldots\ t_n) & \text{if } n > 1. \end{cases}
\end{aligned}$$

REMARK 6.1 An alternative encoding, which is based on a binary sub-division scheme, is given in Hinze (1999). Most generic functions are insensitive to the translation of sums and products. Two notable exceptions are *encode* and *decodes*, for which the binary sub-division scheme is preferable (the linear encoding aggravates the compression rate). □

6.1.2 Specializing generic values

Assume for the sake of example that we want to specialize the generic functions *encode* and *decodes* introduced in Section 1.1.1 to the *List* data type. Recall the types of the generic values (here expressed as type synonyms):

$$\begin{aligned}
&\mathbf{type}\ \mathit{Encode}\ A &&= A \to \mathit{Bin} \\
&\mathbf{type}\ \mathit{Decodes}\ A &&= \mathit{Bin} \to (A, \mathit{Bin}).
\end{aligned}$$

Since $List^\circ$ involves only the type constants '1', '+' and '×' (and the type variables $List$ and A), we can easily specialize *encode* and *decodes* to $List^\circ\ A$: the instances have types $Encode\ (List^\circ\ A)$ and $Decodes\ (List^\circ\ A)$, respectively. However, we require functions of type $Encode\ (List\ A)$ and $Decodes\ (List\ A)$. Now, we already know how to convert between $List^\circ\ A$ and $List\ A$. So it remains to lift $from_{List}$ and to_{List} to functions of type $Encode\ (List\ A) \to Encode\ (List^\circ\ A)$ and $Encode\ (List^\circ\ A) \to Encode\ (List\ A)$. But this lifting is exactly what a mapping function does! In particular, since *Encode* and *Decodes* involve functional types, we can use the embedding-projection maps of Section 5.2.1 for this purpose.

For *mapE* we have to package the two conversion functions into a single value:

$$\begin{array}{lcl} conv_{List} & :: & \forall A\,.\,EP\ (List\ A)\ (List^\circ\ A) \\ conv_{List} & = & ep\{from = from_{List}, to = to_{List}\}. \end{array}$$

Then $encode_{List}$ and $decodes_{List}$ are given by

$$\begin{array}{lcl} encode_{List}\ encodeA & = & to\ (mapE_{Encode}\ conv_{List})\ (encode\langle\!\langle List^\circ\ A\rangle\!\rangle) \\ decodes_{List}\ decodeA & = & to\ (mapE_{Decodes}\ conv_{List})\ (decodes\langle\!\langle List^\circ\ A\rangle\!\rangle). \end{array}$$

Consider the definition of $encode_{List}$. The specialization $encode\langle\!\langle List^\circ\ A\rangle\!\rangle$ yields a function of type $Encode\ (List^\circ\ A)$; the call $to\ (mapE_{Encode}\ conv_{List})$ then converts this function into a value of type $Encode\ (List\ A)$ as desired.

In general, the translation proceeds as follows. For each generic definition we generate the following.

- A type synonym $Poly = Poly\langle\star\rangle$ for the type of the generic value.
- An embedding-projection map, $mapE_{Poly}$, see Section 6.1.3.
- Generic instances for '1', '+', '×' and possibly other primitive types.

For each data type declaration B we generate the following.

- A type synonym, B°, for B's generic representation type, see Section 6.1.1.
- An embedding-projection pair $conv_B$ that converts between $B\ A_1\ \ldots\ A_m$ and $B^\circ\ A_1\ \ldots\ A_m$.

  $$\begin{array}{lcl} conv_B & :: & \forall A_1\ \ldots\ A_m\,.\,EP\ (B\ A_1\ \ldots\ A_m)\ (B^\circ\ A_1\ \ldots\ A_m) \\ conv_B & = & ep\{from = from_B, to = to_B\} \end{array}$$

  The functions $from_B$ and to_B are defined in Section 6.1.1.

An instance of *poly* for type $B :: \mathfrak{B}$ is then given by (using Haskell syntax)

$$\begin{array}{lcl} poly_B & :: & Poly\langle\mathfrak{B}\rangle\ B\ \ldots\ B \\ poly_B\ poly_{A_1}\ \ldots\ poly_{A_m} & = & to\ (mapE_{Poly}\ conv_B\ \ldots\ conv_B)\ (poly\langle\!\langle B^\circ\ A_1\ \ldots\ A_m\rangle\!\rangle). \end{array}$$

If $Poly\langle\mathfrak{B}\rangle\ B\ \ldots\ B$ has a rank of 2 or below, we can express $poly_B$ directly in Haskell. Section 6.1.4 explain the necessary amendments for general rank-n types. Figures 6.1 and 6.2 show several examples of specializations all expressed in Haskell.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{\{- binary encoding -\}}} \\
\mathbf{type}\ \mathit{Encode}\ A & = & A \to \mathit{Bin} \\
\mathit{encode}_1 & :: & \mathit{Encode}\ 1 \\
\mathit{encode}_1 & = & \lambda() \to [\,] \\
\mathit{encode}_+ & :: & \forall A\,.\,\mathit{Encode}\ A \to \forall B\,.\,\mathit{Encode}\ B \to \mathit{Encode}\ (A + B) \\
\mathit{encode}_+\ \mathit{encode}_A\ \mathit{encode}_B & = & \lambda s \to \mathbf{case}\ s\ \mathbf{of}\ \{\,\mathit{inl}\ a \to \mathtt{0} : \mathit{encode}_A\ a; \\
& & \qquad\qquad\qquad\ \ \mathit{inr}\ b \to \mathtt{1} : \mathit{encode}_B\ b\,\} \\
\mathit{encode}_\times & :: & \forall A\,.\,\mathit{Encode}\ A \to \forall B\,.\,\mathit{Encode}\ B \to \mathit{Encode}\ (A \times B) \\
\mathit{encode}_\times\ \mathit{encode}_A\ \mathit{encode}_B & = & \lambda(a, b) \to \mathit{encode}_A\ a \mathbin{+\!\!+} \mathit{encode}_B\ b \\
\mathit{mapE}_{\mathit{Encode}} & :: & \forall A\ B\,.\,\mathit{EP}\ A\ B \to \mathit{EP}\ (\mathit{Encode}\ A)\ (\mathit{Encode}\ B) \\
\mathit{mapE}_{\mathit{Encode}}\ m & = & \mathit{ep}\{\mathit{from} = \lambda h \to h \cdot \mathit{to}\ m, \mathit{to} = \lambda h \to h \cdot \mathit{from}\ m\} \\
\multicolumn{3}{l}{\text{\{- equality -\}}} \\
\mathbf{type}\ \mathit{Equal}\ A_1\ A_2 & = & A_1 \to A_2 \to \mathit{Bool} \\
\mathit{equal}_1 & :: & \mathit{Equal}\ 1\ 1 \\
\mathit{equal}_1 & = & \lambda()\ () \to \mathit{true} \\
\mathit{equal}_+ & :: & \forall A_1\ A_2\,.\,\mathit{Equal}\ A_1\ A_2 \to \forall B_1\ B_2\,.\,\mathit{Equal}\ B_1\ B_2 \\
& & \qquad\qquad \to \mathit{Equal}\ (A_1 + B_1)\ (A_2 + B_2) \\
\mathit{equal}_+\ \mathit{equal}_A\ \mathit{equal}_B & = & \lambda s_1\ s_2 \to \mathbf{case}\ (s_1, s_2)\ \mathbf{of}\ \{(\mathit{inl}\ a_1, \mathit{inl}\ a_2) \to \mathit{equal}_A\ a_1\ a_2; \\
& & \qquad\qquad\qquad\qquad\qquad (\mathit{inl}\ a_1, \mathit{inr}\ b_2) \to \mathit{false}; \\
& & \qquad\qquad\qquad\qquad\qquad (\mathit{inr}\ b_1, \mathit{inl}\ a_2) \to \mathit{false}; \\
& & \qquad\qquad\qquad\qquad\qquad (\mathit{inr}\ b_1, \mathit{inr}\ b_2) \to \mathit{equal}_B\ b_1\ b_2\} \\
\mathit{equal}_\times & :: & \forall A_1\ A_2\,.\,\mathit{Equal}\ A_1\ A_2 \to \forall B_1\ B_2\,.\,\mathit{Equal}\ B_1\ B_2 \\
& & \qquad\qquad \to \mathit{Equal}\ (A_1 \times B_1)\ (A_2 \times B_2) \\
\mathit{equal}_\times\ \mathit{equal}_A\ \mathit{equal}_B & = & \lambda(a_1, b_1)\ (a_2, b_2) \to \mathit{equal}_A\ a_1\ a_2 \wedge \mathit{equal}_B\ b_1\ b_2 \\
\mathit{mapE}_{\mathit{Equal}} & :: & \forall A_1\ B_1\,.\,\mathit{EP}\ A_1\ B_1 \to \forall A_2\ B_2\,.\,\mathit{EP}\ A_2\ B_2 \\
& & \qquad\qquad \to \mathit{EP}\ (\mathit{Equal}\ A_1\ A_2)\ (\mathit{Equal}\ B_1\ B_2) \\
\mathit{mapE}_{\mathit{Equal}}\ m_1\ m_2 & = & \mathit{ep}\{\mathit{from} = \lambda h \to \lambda a_1\ a_2 \to h\ (\mathit{to}\ m_1\ a_1)\ (\mathit{to}\ m_2\ a_2), \\
& & \qquad \mathit{to} = \lambda h \to \lambda b_1\ b_2 \to h\ (\mathit{from}\ m_1\ b_1)\ (\mathit{from}\ m_2\ b_2)\} \\
\multicolumn{3}{l}{\text{\{- generic representation types -\}}} \\
\mathbf{type}\ \mathit{Maybe}^\circ\ A & = & 1 + A \\
\mathit{from}_{\mathit{Maybe}} & :: & \forall A\,.\,\mathit{Maybe}\ A \to \mathit{Maybe}^\circ\ A \\
\mathit{from}_{\mathit{Maybe}}\ \mathit{nothing} & = & \mathit{inl}\ () \\
\mathit{from}_{\mathit{Maybe}}\ (\mathit{just}\ a) & = & \mathit{inr}\ a \\
\mathit{to}_{\mathit{Maybe}} & :: & \forall A\,.\,\mathit{Maybe}^\circ\ A \to \mathit{Maybe}\ A \\
\mathit{to}_{\mathit{Maybe}}\ (\mathit{inl}\ ()) & = & \mathit{nothing} \\
\mathit{to}_{\mathit{Maybe}}\ (\mathit{inr}\ a) & = & \mathit{just}\ a \\
\mathit{conv}_{\mathit{Maybe}} & :: & \forall A\,.\,\mathit{EP}\ (\mathit{Maybe}\ A)\ (\mathit{Maybe}^\circ\ A) \\
\mathit{conv}_{\mathit{Maybe}} & = & \mathit{ep}\{\mathit{from} = \mathit{from}_{\mathit{Maybe}}, \mathit{to} = \mathit{to}_{\mathit{Maybe}}\}
\end{array}
$$

Figure 6.1: Specializing generic values in Haskell (part 1).

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{List}^{\circ}\ A & = & 1 + A \times \mathit{List}\ A \\
\mathit{from}_{\mathit{List}} & :: & \forall A\,.\,\mathit{List}\ A \rightarrow \mathit{List}^{\circ}\ A \\
\mathit{from}_{\mathit{List}}\ [\,] & = & \mathit{inl}\ () \\
\mathit{from}_{\mathit{List}}\ (a : as) & = & \mathit{inr}\ (a, as) \\
\mathit{to}_{\mathit{List}} & :: & \forall A\,.\,\mathit{List}^{\circ}\ A \rightarrow \mathit{List}\ A \\
\mathit{to}_{\mathit{List}}\ (\mathit{inl}\ ()) & = & [\,] \\
\mathit{to}_{\mathit{List}}\ (\mathit{inr}\ (a, as)) & = & a : as \\
\mathit{conv}_{\mathit{List}} & :: & \forall A\,.\,\mathit{EP}\ (\mathit{List}\ A)\ (\mathit{List}^{\circ}\ A) \\
\mathit{conv}_{\mathit{List}} & = & \mathit{ep}\{\mathit{from} = \mathit{from}_{\mathit{List}}, \mathit{to} = \mathit{to}_{\mathit{List}}\} \\
\textbf{type}\ \mathit{GRose}^{\circ}\ F\ A & = & A \times F\ (\mathit{GRose}\ F\ A) \\
\mathit{from}_{\mathit{GRose}} & :: & \forall F\ A\,.\,\mathit{GRose}\ F\ A \rightarrow \mathit{GRose}^{\circ}\ F\ A \\
\mathit{from}_{\mathit{GRose}}\ (\mathit{gbranch}\ a\ ts) & = & (a, ts) \\
\mathit{to}_{\mathit{GRose}} & :: & \forall F\ A\,.\,\mathit{GRose}^{\circ}\ F\ A \rightarrow \mathit{GRose}\ F\ A \\
\mathit{to}_{\mathit{GRose}}\ (a, ts) & = & \mathit{gbranch}\ a\ ts \\
\mathit{conv}_{\mathit{GRose}} & :: & \forall F\ A\,.\,\mathit{EP}\ (\mathit{GRose}\ F\ A)\ (\mathit{GRose}^{\circ}\ F\ A) \\
\mathit{conv}_{\mathit{GRose}} & = & \mathit{ep}\{\mathit{from} = \mathit{from}_{\mathit{GRose}}, \mathit{to} = \mathit{to}_{\mathit{GRose}}\}
\end{array}
$$

{- specializing binary encoding -}

$$
\begin{array}{lcl}
\mathit{encode}_{\mathit{Maybe}} & :: & \forall A\,.\,\mathit{Encode}\ A \rightarrow \mathit{Encode}\ (\mathit{Maybe}\ A) \\
\mathit{encode}_{\mathit{Maybe}}\ \mathit{encode}_A & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Encode}}\ \mathit{conv}_{\mathit{Maybe}})\ (\mathit{encode}_{+}\ \mathit{encode}_1\ \mathit{encode}_A) \\
\mathit{encode}_{\mathit{List}} & :: & \forall A\,.\,\mathit{Encode}\ A \rightarrow \mathit{Encode}\ (\mathit{List}\ A) \\
\mathit{encode}_{\mathit{List}}\ \mathit{encode}_A & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Encode}}\ \mathit{conv}_{\mathit{List}})\ (\\
 & & \quad \mathit{encode}_{+}\ \mathit{encode}_1\ (\mathit{encode}_{\times}\ \mathit{encode}_A\ (\mathit{encode}_{\mathit{List}}\ \mathit{encode}_A))) \\
\mathit{encode}_{\mathit{GRose}} & :: & \forall F\,.\,(\forall B\,.\,\mathit{Encode}\ B \rightarrow \mathit{Encode}\ (F\ B)) \\
 & & \qquad \rightarrow (\forall A\,.\,\mathit{Encode}\ A \rightarrow \mathit{Encode}\ (\mathit{GRose}\ F\ A)) \\
\multicolumn{3}{l}{\mathit{encode}_{\mathit{GRose}}\ \mathit{encode}_F\ \mathit{encode}_A} \\
 & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Encode}}\ \mathit{conv}_{\mathit{GRose}})\ (\\
 & & \quad \mathit{encode}_{\times}\ \mathit{encode}_A\ (\mathit{encode}_F\ (\mathit{encode}_{\mathit{GRose}}\ \mathit{encode}_F\ \mathit{encode}_A)))
\end{array}
$$

{- specializing equality -}

$$
\begin{array}{lcl}
\mathit{equal}_{\mathit{Maybe}} & :: & \forall A_1\ A_2\,.\,\mathit{Equal}\ A_1\ A_2 \rightarrow \mathit{Equal}\ (\mathit{Maybe}\ A_1)\ (\mathit{Maybe}\ A_2) \\
\mathit{equal}_{\mathit{Maybe}}\ \mathit{equal}_A & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Equal}}\ \mathit{conv}_{\mathit{Maybe}}\ \mathit{conv}_{\mathit{Maybe}})\ (\mathit{equal}_{+}\ \mathit{equal}_1\ \mathit{equal}_A) \\
\mathit{equal}_{\mathit{List}} & :: & \forall A_1\ A_2\,.\,\mathit{Equal}\ A_1\ A_2 \rightarrow \mathit{Equal}\ (\mathit{List}\ A_1)\ (\mathit{List}\ A_2) \\
\mathit{equal}_{\mathit{List}}\ \mathit{equal}_A & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Equal}}\ \mathit{conv}_{\mathit{List}}\ \mathit{conv}_{\mathit{List}})\ (\\
 & & \quad \mathit{equal}_{+}\ \mathit{equal}_1\ (\mathit{equal}_{\times}\ \mathit{equal}_A\ (\mathit{equal}_{\mathit{List}}\ \mathit{equal}_A))) \\
\mathit{equal}_{\mathit{GRose}} & :: & \forall F_1\ F_2\,.\,(\forall B_1\ B_2\,.\,\mathit{Equal}\ B_1\ B_2 \rightarrow \mathit{Equal}\ (F_1\ B_1)\ (F_2\ B_2)) \\
 & & \qquad\quad \rightarrow (\forall A_1\ A_2\,.\,\mathit{Equal}\ A_1\ A_2 \\
 & & \qquad\qquad\qquad\qquad \rightarrow \mathit{Equal}\ (\mathit{GRose}\ F_1\ A_1)\ (\mathit{GRose}\ F_2\ A_2)) \\
\multicolumn{3}{l}{\mathit{equal}_{\mathit{GRose}}\ \mathit{equal}_F\ \mathit{equal}_A} \\
 & = & \mathit{to}\ (\mathit{mapE}_{\mathit{Equal}}\ \mathit{conv}_{\mathit{GRose}}\ \mathit{conv}_{\mathit{GRose}})\ (\\
 & & \quad \mathit{equal}_{\times}\ \mathit{equal}_A\ (\mathit{equal}_F\ (\mathit{equal}_{\mathit{GRose}}\ \mathit{equal}_F\ \mathit{equal}_A)))
\end{array}
$$

Figure 6.2: Specializing generic values in Haskell (part 2).

6.1.3 Generating embedding-projection maps

We are in a peculiar situation: in order to specialize a generic value *poly* to some data type B, we have to specialize another generic value, namely, *mapE* to *poly*'s type *Poly*. This works fine if *Poly* like *Encode* only involves primitive types. So let us make this assumption for the moment. Here is a version of *mapE* tailored to Haskell's set of primitive types:

$$\begin{array}{lcl}
mapE\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & :: & MapE\langle\mathfrak{T}\rangle\ T\ T \\
mapE\langle\!\langle Char\rangle\!\rangle & = & idE \\
mapE\langle\!\langle Int\rangle\!\rangle & = & idE \\
mapE\langle\!\langle \rightarrow\rangle\!\rangle\ mA\ mB & = & ep\{from = to\ mA \rightarrow from\ mB, to = from\ mA \rightarrow to\ mB\} \\
mapE\langle\!\langle IO\rangle\!\rangle\ mA & = & ep\{from = fmap\ (from\ mA), to = fmap\ (to\ mA)\}.
\end{array}$$

Note that in the last equation *mapE* falls back on an 'ordinary' mapping function. In fact, we can alternatively define

$$mapE\langle\!\langle IO\rangle\!\rangle \quad = \quad liftE$$

where

$$\begin{array}{lcl}
liftE & :: & \forall F\,.\,(Functor\ F) \Rightarrow \forall A\ A^\circ\,.\,EP\ A\ A^\circ \rightarrow EP\ (F\ A)\ (F\ A^\circ) \\
liftE\ mA & = & ep\{from = fmap\ (from\ mA), to = fmap\ (to\ mA)\}.
\end{array}$$

Now, the $Poly :: \_POLY$ instance of *mapE* is given by

$$\begin{array}{lcl}
mapE_{Poly} & :: & MapE\langle_POLY\rangle\ Poly\ Poly \\
mapE_{Poly}\ mapE_{A_1}\ \ldots\ mapE_{A_k} & = & mapE\langle\!\langle Poly\ A_1\ \ldots\ A_k\rangle\!\rangle\varrho.
\end{array}$$

where $\varrho = (A_1 := mapE_{A_1}, \ldots, A_k := mapE_{A_k})$ is an environment mapping type variables to terms. We use an explicit environment (actually, for the first time) in order to deal with polymorphic types. Recall that the specialization of generic values as described in Section 3.1.3 does not work for polymorphic types. However, we allow polymorphic types to occur in the type signature of a generic value. Now, the extension of *mapE* to arbitrary Haskell type terms is given by

$$\begin{array}{lcl}
mapE\langle\!\langle C\rangle\!\rangle\varrho & = & mapE\langle\!\langle C\rangle\!\rangle \\
mapE\langle\!\langle A\rangle\!\rangle\varrho & = & \varrho(A) \\
mapE\langle\!\langle T\ U\rangle\!\rangle\varrho & = & (mapE\langle\!\langle T\rangle\!\rangle\varrho)\ (mapE\langle\!\langle U\rangle\!\rangle\varrho) \\
mapE\langle\!\langle \forall A :: \star\,.\,T\rangle\!\rangle\varrho & = & mapE\langle\!\langle T\rangle\!\rangle\varrho(A := idE) \\
mapE\langle\!\langle \forall F :: \star \rightarrow \star\,.\,(Functor\ F) \Rightarrow T\rangle\!\rangle\varrho & = & mapE\langle\!\langle T\rangle\!\rangle\varrho(F := liftE).
\end{array}$$

Two remarks are in order.

Haskell has neither type abstractions nor an explicit recursion operator, so these cases can be omitted from the definition.

Unfortunately, we cannot deal with polymorphic types in general. Consider, for instance, the type $Poly\ A = \forall F\,.\,F\ A \rightarrow F\ A$. There is no mapping function that works uniformly for all F. For that reason we have to restrict F to instances of *Functor* so that we can use the overloaded *liftE* function. For polymorphic types where the type variable ranges over types of kind $\star$ things are simpler: since the mapping function for a manifest type is always the identity, we can use *idE*.

Now, what happens if *Poly* involves a user-defined data type, say B? In this case we have to specialize *mapE* to B. It seems that we are trapped in a vicious circle. To break the spell we have to implement *mapE* for the B data type 'by hand'. Fortunately $mapE$ is very well-behaved, so the code generation is relatively straightforward. The embedding-projection map for the data type $B :: \mathfrak{B}$

$$\textbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}$$

is given by

$$\begin{array}{lcl}
mapE_B & :: & MapE\langle\mathfrak{B}\rangle\ B\ B \\
mapE_B\ mapE_{A_1}\ \ldots\ mapE_{A_m} & & \\
 & = & ep\{from = from_B, to = to_B\} \\
\quad \textbf{where} & & \\
\quad from_B\ (k_1\ x_{11}\ \ldots\ x_{1m_1}) & = & k_1\ (from\ (mapE\langle\!\langle T_{11}\rangle\!\rangle\varrho)\ x_{11})\ \ldots\ (from\ (mapE\langle\!\langle T_{1m_1}\rangle\!\rangle\varrho)\ x_{1m_1}) \\
\quad \ldots & & \\
\quad from_B\ (k_n\ x_{n1}\ \ldots\ x_{nm_n}) & = & k_n\ (from\ (mapE\langle\!\langle T_{n1}\rangle\!\rangle\varrho)\ x_{n1})\ \ldots\ (from\ (mapE\langle\!\langle T_{nm_n}\rangle\!\rangle\varrho)\ x_{nm_n}) \\
\quad to_B\ (k_1\ x_{11}\ \ldots\ x_{1m_1}) & = & k_1\ (to\ (mapE\langle\!\langle T_{11}\rangle\!\rangle\varrho)\ x_{11})\ \ldots\ (to\ (mapE\langle\!\langle T_{1m_1}\rangle\!\rangle\varrho)\ x_{1m_1}) \\
\quad \ldots & & \\
\quad to_B\ (k_n\ x_{n1}\ \ldots\ x_{nm_n}) & = & k_n\ (to\ (mapE\langle\!\langle T_{n1}\rangle\!\rangle\varrho)\ x_{n1})\ \ldots\ (to\ (mapE\langle\!\langle T_{nm_n}\rangle\!\rangle\varrho)\ x_{nm_n})
\end{array}$$

where $\varrho = (A_1 := mapE_{A_1}, \ldots, A_m := mapE_{A_m})$. For example, for $Encode$ and $Decodes$ we obtain

$$\begin{array}{lcl}
mapE_{Encode} & :: & \forall A\ A^\circ\,.\,EP\ A\ A^\circ \to EP\ (Encode\ A)\ (Encode\ A^\circ) \\
mapE_{Encode}\ mapE_A & = & mapE_{\to}\ mapE_A\ idE \\
mapE_{Decodes} & :: & \forall A\ A^\circ\,.\,EP\ A\ A^\circ \to EP\ (Decodes\ A)\ (Decodes\ A^\circ) \\
mapE_{Decodes}\ mapE_A & = & mapE_{\to}\ idE\ (mapE_{(,)}\ mapE_A\ idE)
\end{array}$$

where $mapE_{(,)}$ is generated according to the scheme above:

$$\begin{array}{lcl}
mapE_{(,)} & :: & \forall A\ A^\circ\,.\,EP\ A\ A^\circ \to \forall B\ B^\circ\,.\,EP\ B\ B^\circ \to EP\ (A,B)\ (A^\circ,B^\circ) \\
mapE_{(,)}\ mapE_A\ mapE_B & = & ep\{from = from_{(,)}, to = to_{(,)}\} \\
\quad \textbf{where}\ from_{(,)}\ (a,b) & = & (from\ mapE_A\ a, from\ mapE_B\ b) \\
\qquad\qquad to_{(,)}\ (a,b) & = & (to\ mapE_A\ a, to\ mapE_B\ b).
\end{array}$$

6.1.4 Encoding rank-n types

The translation described in Section 6.1.2 can be used as a source-to-source translation provided the types of the functions involved have a rank of 2 or below. In this section we close the gap and show how to encode rank-n types using 'wrapper' data types with polymorphic fields. Note that polymorphic fields are an extension to Haskell 98 implemented in GHC, HBC and Hugs. Now, the basic idea is very simple: instead of passing a polymorphic value directly as an argument we pass a 'box' that contains the value as the single component.

Assume for the sake of example that only rank-0 or rank-1 type signatures are admissible and consider specializing $encode$ to $GRose$. In this case we have to pass the first argument of $encode_{GRose}$ as a boxed value. A suitable data type for this purpose is

$$\textbf{newtype}\ Boxed_{\star\to\star}\ F \quad = \quad box_{\star\to\star}\{unbox_{\star\to\star} :: \forall A\,.\,Encode\ A \to Encode\ (F\ A)\}.$$

The instance $encode_{GRose}$ then takes the following form

$$\begin{array}{l}
encode_{GRose}\ \ ::\ \ \forall F\ A\,.\,Boxed_{\star\to\star}\ F \to Encode\ A \to Encode\ (GRose\ F\ A) \\
encode_{GRose}\ encode_F\ encode_A \\
\qquad\qquad = \ to\ (mapE_{Encode}\ conv_{GRose})\ (\\
\qquad\qquad\qquad encode_{\times}\ encode_A\ (unbox_{\star\to\star}\ encode_F\ (encode_{GRose}\ encode_F\ encode_A))\ ).
\end{array}$$

Note that we have to unbox the boxed value $encode_F$ before we can apply it.

Now, how can we introduce the wrapper data types $Boxed_{\mathfrak{T}}$ and the conversion functions $unbox_{\mathfrak{T}}$ and $box_{\mathfrak{T}}$ in a systematic way? It appears that this is most easily accomplished by introducing a new constructor on the *kind level*: we use $\boxdot\ \mathfrak{T}$ to indicate that the corresponding instance must be boxed. At first, it may seem bizarre that this information is introduced on the kind level. But recall that the type of an instance is defined by induction on the structure of kinds, that is, the kind of the type T determines the type of the instance $poly\langle\!\langle T\rangle\!\rangle$. For instance, if we specialize $poly$ to $GRose$ we assign $poly\langle\!\langle GRose\rangle\!\rangle$ the type $Poly\langle \boxdot\ (\star \to \star) \to \star \to \star\rangle$ indicating that the first argument of $poly\langle\!\langle GRose\rangle\!\rangle$ must be boxed.

Now, we extend the language of kind terms as follows.

$$
\begin{array}{rcll}
\mathfrak{T}, \mathfrak{U} \in \mathfrak{Kind} & ::= & \star & \text{kind of types} \\
& | & \mathfrak{T} \times \mathfrak{U} & \text{product kind} \\
& | & \mathfrak{T} \to \mathfrak{U} & \text{function kind} \\
& | & \boxdot\ \mathfrak{T} & \text{boxed kind}
\end{array}
$$

Furthermore, we introduce two kinding rules that allow to introduce and to eliminate boxed kinds.

$$
\frac{T :: \mathfrak{T}}{T :: \boxdot\ \mathfrak{T}}\ (\textsc{t-}\boxdot\textsc{-intro}) \qquad \frac{T :: \boxdot\ \mathfrak{T}}{T :: \mathfrak{T}}\ (\textsc{t-}\boxdot\textsc{-elim})
$$

Note that we do not introduce any constructors on the type level, so just think of $\boxdot\ \mathfrak{T}$ and $\mathfrak{T}$ as two isomorphic kinds without the need to explicitly coerce to and fro.

Turning to the specialization we have to extend the definition of $Poly\langle -\rangle$.

$$
\begin{array}{lcl}
\textbf{type}\ Poly\langle\star\rangle\ T_1\ \ldots\ T_n & = & Poly\ T_1\ \ldots\ T_n \\
\textbf{type}\ Poly\langle\boxdot\ \mathfrak{T}\rangle\ T_1\ \ldots\ T_n & = & Boxed_{\mathfrak{T}}\ T_1\ \ldots\ T_n \\
\textbf{type}\ Poly\langle\mathfrak{A} \to \mathfrak{B}\rangle\ T_1\ \ldots\ T_n & = & \forall A_1\ \ldots\ A_n\,.\,Poly\langle\mathfrak{A}\rangle\ A_1\ \ldots\ A_n \\
& & \qquad \to Poly\langle\mathfrak{B}\rangle\ (T_1\ A_1)\ \ldots\ (T_n\ A_n) \\
\textbf{newtype}\ Boxed_{\mathfrak{T}}\ T_1\ \ldots\ T_n & = & box_{\mathfrak{T}}\{\,unbox_{\mathfrak{T}} :: Poly\langle\mathfrak{T}\rangle\ T_1\ \ldots\ T_n\}
\end{array}
$$

Note that $Boxed_{\mathfrak{T}}$ and $Poly\langle\mathfrak{T}\rangle$ are isomorphic type constructors. We can use $box_{\mathfrak{T}}$ and $unbox_{\mathfrak{T}}$ to convert to and fro. Next, we have to extend the definition of $poly\langle\!\langle -\rangle\!\rangle$. Recall that $poly\langle\!\langle -\rangle\!\rangle$ is defined by induction on the structure of the kinding derivations (this is why we do not need coercions on the type level).

$$
\begin{array}{lcl}
poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle & = & unbox_{\mathfrak{T}}\ (poly\langle\!\langle T :: \boxdot\ \mathfrak{T}\rangle\!\rangle) \\
poly\langle\!\langle T :: \boxdot\ \mathfrak{T}\rangle\!\rangle & = & box_{\mathfrak{T}}\ (poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle)
\end{array}
$$

Using boxed kinds we can now easily tailor the code generation towards a particular target language. Assume, as before, that the target language only admits rank-0 and rank-1 type signatures (but, of course, it must offer local universal quantification in data types) and that we want to specialize $poly\langle\!\langle T :: \mathfrak{T}\rangle\!\rangle$. In this case we specialize $poly\langle\!\langle T :: wrap_1\ \mathfrak{T}\rangle\!\rangle$ where $wrap_1$ is given by

$$
\begin{array}{lcl}
wrap_1 & :: & \Box \to \Box \\
wrap_1(\star) & = & \star \\
wrap_1(\mathfrak{T} \to \mathfrak{U}) & & \\
\quad |\ order(\mathfrak{T}) \geqslant 1 & = & \boxdot\ (wrap_1\ \mathfrak{T}) \to wrap_1\ \mathfrak{U} \\
\quad |\ otherwise & = & \mathfrak{T} \to wrap_1\ \mathfrak{U}.
\end{array}
$$

The function $wrap_1$ introduces boxed kinds for higher-order type arguments, for example, $wrap_1\ ((\star \to \star) \to \star \to \star) = \boxdot\ (\star \to \star) \to \star \to \star$. Now, reconsider the definition of $encode_{GRose}$ and note the first argument, $encode_F$, appears twice on the right-hand side. Furthermore, note that only the first occurrence is unboxed. The kinding derivation of $GRose^\circ\ F\ A :: \star$ with $F :: \boxdot\ (\star \to \star)$ and $A :: \star$ shows why.

$$\begin{array}{rll}
(1) & \vdash (\times)\ A\ (F\ (GRose^\circ\ F\ A)) :: \star & \text{T-}\to\text{-ELIM}(2,\ 3) \\
(2) & \vdash (\times)\ A :: \star \to \star & \text{T-}\to\text{-ELIM}(4,\ 5) \\
(3) & \vdash F\ (GRose^\circ\ F\ A) :: \star & \text{T-}\to\text{-ELIM}(6,\ 7) \\
(4) & \vdash (\times) :: \star \to (\star \to \star) & \text{T-CONST} \\
(5) & \vdash A :: \star & \text{T-VAR} \\
(6) & \vdash F :: \star \to \star & \text{T-}\boxdot\text{-ELIM}(8) \\
(7) & \vdash GRose^\circ\ F\ A :: \star & \text{T-}\to\text{-ELIM}(9,\ 10) \\
(8) & \vdash F :: \boxdot\ (\star \to \star) & \text{T-VAR} \\
(9) & \vdash GRose^\circ\ F :: \star \to \star & \text{T-}\to\text{-ELIM}(11,\ 12) \\
(10) & \vdash A :: \star & \text{T-VAR} \\
(11) & \vdash GRose^\circ :: \boxdot\ (\star \to \star) \to (\star \to \star) & \text{T-VAR} \\
(12) & \vdash F :: \boxdot\ (\star \to \star) & \text{T-VAR}
\end{array}$$

In line (6) we require $F :: \star \to \star$ but F has kind $\boxdot\ (\star \to \star)$, so that we have to invoke ($\text{T-}\boxdot\text{-ELIM}$). Consequently, we obtain

$$\begin{array}{rcl}
encode\langle\!\langle F :: \star \to \star \rangle\!\rangle & = & unbox_{\star\to\star}\ (encode\langle\!\langle F :: \boxdot\ (\star \to \star)\rangle\!\rangle) \\
 & = & unbox_{\star\to\star}\ encode_F\,.
\end{array}$$

By contrast, in line (12) we require $F :: \boxdot\ (\star \to \star)$. Since F has exactly this kind, we obtain $encode\langle\!\langle F :: \boxdot\ (\star \to \star)\rangle\!\rangle = encode_F$.

Now, GHC, HBC and Hugs also offer rank-2 type signatures. In this case, there is no need to box first-order kinded type arguments. The wrapper function $wrap_2$ takes this into account:

$$\begin{array}{lcl}
wrap_2 & :: & \square \to \square \\
wrap_2(\star) & = & \star \\
wrap_2(\mathfrak{T} \to \mathfrak{U}) & & \\
\quad \mid order(\mathfrak{T}) \geqslant 2 & = & \boxdot\ (wrap_1\ \mathfrak{T}) \to wrap_2\ \mathfrak{U} \\
\quad \mid otherwise & = & \mathfrak{T} \to wrap_2\ \mathfrak{U}.
\end{array}$$

Note that the argument of '$\boxdot$' is boxed using $wrap_1$ (not $wrap_2$) since arguments of value constructors may only have rank-1 type signatures.

6.2 Extensions

This section discusses two extensions that make generic definitions more useful in practice.

6.2.1 Ad-hoc definitions

A generic function solves a problem in a uniform way for all types. Sometimes it is, however, desirable to use a different approach for some data types. Consider, for instance, the function *encode* instantiated to lists over some base type. To encode the structure of an n-element list $n + 1$ bits are used. For large lists this is clearly wasteful. A more space-efficient scheme stores the length of the list in a header followed by the encodings of the elements. We can specify this compression scheme for lists using a so-called *ad-hoc definition*.

$$encode\langle List\ A\rangle\ as \quad = \quad encodeInt\ (sizeList\ as) \mathbin{+\!\!+} encodeListBin\ (mapList\ (encode\langle A\rangle)\ as).$$

Ad-hoc definitions specify exceptions to the general rule and may be given for all predefined and for all user-defined data types. Note that this ability is absolutely crucial to support abstract data types. For example, a set may be represented as a balanced tree in more than one way, and equality must take account of this fact. Simply using a generic equality function would take equality of *representations*, which is simply wrong in this case.

In general, generic definitions can be handled very much like class- and instance declarations. The type signature of a generic definition together with the equations for '1', '+', and '×' plays the rôle of a class definition. Ad-hoc definitions are akin to instance declarations. This suggests, for instance, that in a concrete implementation ad-hoc definitions should be allowed to be spread over several modules. This is vital, because a data type might not even be defined in the scope where the generic value is declared.

6.2.2 Constructor names and record labels

Generic definitions are defined by induction on the *structure* of types. Annoyingly, this is not quite enough. Consider, for example, the method *showsPrec* of the standard Haskell class *Show*. To be able to give a generic definition for *showsPrec*, the *names* of the constructors, and their fixities, must be made accessible.

To this end we provide an additional type pattern, of the form c **of** A where c is a *value* variable of type *ConDescr* and A is a *type* variable. The type *ConDescr* is a new primitive type that comprises all constructor names. To manipulate constructor names the following operations among others can be used — for an exhaustive list see Hinze (1999).

$$\begin{array}{lcll}
\mathbf{data}\ ConDescr & & & \text{-- abstract} \\
\mathbf{data}\ Fixity & = & Nofix \mid Infix\ Int \mid Infixl\ Int \mid Infixr\ Int & \\
conName & :: & ConDescr \to String & \text{-- primitive} \\
conArity & :: & ConDescr \to Int & \text{-- primitive} \\
conFixity & :: & ConDescr \to Fixity & \text{-- primitive}
\end{array}$$

Using *conName* and *conArity* we can implement a simple variant of the *showsPrec* function — for a full-fledged version see Hinze (1999). The generic function $showPrec\langle T\rangle\ d\ t$ takes a precedence level d (a value from 0 to 10), a value t of type T and returns a *String*.

$$\begin{array}{lcl}
showPrec\langle T :: \star\rangle & :: & Int \to T \to String \\
showPrec\langle Char\rangle\ d\ c & = & showChar\ c \\
showPrec\langle Int\rangle\ d\ i & = & showInt\ i \\
showPrec\langle A + B\rangle\ d\ (inl\ a) & = & showPrec\langle A\rangle\ d\ a \\
showPrec\langle A + B\rangle\ d\ (inr\ b) & = & showPrec\langle B\rangle\ d\ b \\
showPrec\langle c\ \mathbf{of}\ A\rangle\ d\ a & & \\
\quad \mid conArity\ c == 0 & = & conName\ c \\
\quad \mid otherwise & = & showParen\ (d \geqslant 10)\ (conName\ c \mathbin{+\!\!+} \text{"␣"} \mathbin{+\!\!+} showPrec\langle A\rangle\ 10\ a) \\
showPrec\langle A \times B\rangle\ d\ (a, b) & = & showPrec\langle A\rangle\ d\ a \mathbin{+\!\!+} \text{"␣"} \mathbin{+\!\!+} showPrec\langle B\rangle\ d\ b
\end{array}$$

The third and the fourth equation discard the binary constructors *inl* and *inr*. They are not required since the constructor names are accessible via the type pattern c **of** A. If the constructor is nullary, its string representation is emitted. Otherwise, the constructor name is printed followed by a space followed by the representation of its arguments. If the precedence level is 10, the output is additionally parenthesized. The last equation applies if a constructor has more than one component. In this case the components are separated by a space.

It should be noted that constructor names appear only on the type level; they have no counterpart on the value level as value constructors are encoded using *inl* and *inr*. If a generic definition does not include a case for the type pattern c **of** A, then we tacitly assume that $poly\langle c \textbf{ of } A\rangle = poly\langle A\rangle$. Now, why does the type c **of** A incorporate information about c? One might suspect that it is sufficient to supply this information on the value level. Doing so would work for *show*, but would fail for *read*:

$$\begin{array}{lcl} read\langle T :: \star\rangle & :: & String \to [(T, String)] \\ \ldots & & \\ read\langle c \textbf{ of } A\rangle\ s & = & [(x, s_3) \mid (s_1, s_2) \leftarrow lex\ s, s_1 == conName\ c, \\ & & \qquad\qquad (x, s_3) \leftarrow read\langle A\rangle\ s_2]. \end{array}$$

The important point is that *read produces* (not consumes) the value, and yet it requires access to the constructor name.

Haskell allows the programmer to assign labels to the components of a constructor, and these, too, are needed by *read* and *show*. For the purpose of presentation, however, we choose to ignore field names. In fact, they can be handled completely analogously to constructor names, see Hinze (1999).

It remains to extend the definition of generic representation types to include c **of** A patterns: for

$$\textbf{data}\ B\ A_1\ \ldots\ A_m \quad = \quad k_1\ T_{11}\ \ldots\ T_{1m_1} \mid \cdots \mid k_n\ T_{n1}\ \ldots\ T_{nm_n}$$

we generate:

$$\textbf{type}\ B^\circ\ A_1\ \ldots\ A_m \quad = \quad \Sigma\,(descr_{k_1} \textbf{ of } (\Pi\ T_{11}\ \ldots\ T_{1m_1}))\ \cdots\ (descr_{k_n} \textbf{ of } (\Pi\ T_{n1}\ \ldots\ T_{nm_n}))$$

where $descr_{k_1}, \ldots, descr_{k_n}$ are elements of type *ConDescr*. In fact, for each constructor in a data type declaration, we produce a value of type *ConDescr* that gives information about the constructor:

$$\begin{array}{lcl} \textbf{data}\ ConDescr & = & ConDescr\{\, conName :: String, \\ & & \qquad\qquad\quad conArity :: Int, \\ & & \qquad\qquad\quad conFixity :: Fixity\,\}. \end{array}$$

As an example, for the *List* data type we generate:

$$\begin{array}{lcl} descr_{nil}, descr_{cons} & :: & ConDescr \\ descr_{nil} & = & ConDescr\ \texttt{"Nil"}\ 0\ Nofix \\ descr_{cons} & = & ConDescr\ \texttt{"Cons"}\ 2\ Nofix. \end{array}$$

Let us conclude the section by giving a further example of a generic definition that uses c **of** A patterns. The generic function $layn\langle T\rangle\ \mathit{off}\ t$ displays the value t

of type T in a tree-like fashion.

$$
\begin{array}{lcl}
\mathit{layn}\langle T :: \star\rangle & :: & \mathit{Int} \to T \to \mathit{String} \\
\mathit{layn}\langle 1\rangle\ \mathit{off}\ () & = & \texttt{""} \\
\mathit{layn}\langle \mathit{Int}\rangle\ \mathit{off}\ i & = & \mathit{line}\ \mathit{off}\ (\mathit{showInt}\ i) \\
\mathit{layn}\langle A + B\rangle\ \mathit{off}\ (\mathit{inl}\ a) & = & \mathit{layn}\langle A\rangle\ \mathit{off}\ a \\
\mathit{layn}\langle A + B\rangle\ \mathit{off}\ (\mathit{inr}\ b) & = & \mathit{layn}\langle B\rangle\ \mathit{off}\ b \\
\mathit{layn}\langle c\ \mathbf{of}\ A\rangle\ \mathit{off}\ a & = & \mathit{line}\ \mathit{off}\ (\mathit{conName}\ c) +\!\!+\ \mathit{layn}\langle A\rangle\ (\mathit{off} + 2)\ a \\
\mathit{layn}\langle A \times B\rangle\ \mathit{off}\ (a, b) & = & \mathit{layn}\langle A\rangle\ \mathit{off}\ a +\!\!+ \texttt{"\textbackslash n"} +\!\!+\ \mathit{layn}\langle B\rangle\ \mathit{off}\ b \\
\mathit{line} & :: & \mathit{Int} \to \mathit{String} \to \mathit{String} \\
\mathit{line}\ \mathit{off}\ s & = & \mathit{replicate}\ \mathit{off}\ \texttt{'␣'} +\!\!+\ s +\!\!+ \texttt{"\textbackslash n"}
\end{array}
$$

A constructed value of the form $k\ t_1\ \ldots\ t_n$ is displayed as follows.

$$
\begin{array}{l}
\texttt{␣␣}\cdots\texttt{␣␣}k \\
\texttt{␣␣}\cdots\texttt{␣␣␣␣}t_1 \\
\texttt{␣␣}\cdots\texttt{␣␣␣␣}\cdots \\
\texttt{␣␣}\cdots\texttt{␣␣␣␣}t_n
\end{array}
$$

The constructor name k is printed on a separate line using an offset of *off* spaces; its components $t_1, \ldots, t_n$ are recursively displayed using an offset of $\mathit{off} + 2$ spaces.

References

Amadio, R., K. B. Bruce, and G. Longo (1986, June). The finitary projection model for second order lambda calculus and solutions to higher order domain equations. In *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pp. 122–130. IEEE Computer Society.

Appel, A. W. and M. J. R. Goncalves (1993, February). Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department.

Augustsson, L. (1998). *The HBC compiler.* Available from `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`.

Augustsson, L. (1999, January). Cayenne – a language with dependent types. *SIGPLAN Notices 34*(1), 239–250.

Backhouse, R., P. Jansson, J. Jeuring, and L. Meertens (1999). Generic Programming — An Introduction —. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira (Eds.), *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, Volume 1608 of *Lecture Notes in Computer Science*, pp. 28–115. Berlin: Springer-Verlag.

Barendregt, H. P. (1984). *The Lambda Calculus — Its Syntax and Semantics* (revised ed.). North-Holland, Amsterdam New York Oxford.

Bird, R. (1998). *Introduction to Functional Programming using Haskell* (2nd ed.). London: Prentice Hall Europe.

Bird, R. and O. de Moor (1997). *Algebra of Programming.* London: Prentice Hall Europe.

Bird, R., O. de Moor, and P. Hoogendijk (1996, January). Generic functional programming with types and relations. *Journal of Functional Programming 6*(1), 1–28.

Bird, R. and L. Meertens (1998, June). Nested datatypes. In J. Jeuring (Ed.), *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, Volume 1422 of *Lecture Notes in Computer Science*, pp. 52–67. Springer-Verlag.

Bird, R. and R. Paterson (1999). Generalised folds for nested datatypes. *Formal Aspects of Computing 11*(2), 200–222.

Bird, R. S. (1984, October). The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems 6*(4), 487–504.

Cockett, R. and T. Fukushima (1992, June). About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary.

Connelly, R. H. and F. L. Morris (1995, September). A generalization of the trie data structure. *Mathematical Structures in Computer Science 5*(3), 381–418.

Courcelle, B. (1983, March). Fundamental properties of infinite trees. *Theoretical Computer Science 25*(2), 95–169.

Crary, K., S. Weirich, and G. Morrisett (1999). Intensional polymorphism in type-erasure semantics. *ACM SIGPLAN Notices 34*(1), 301–312.

Danvy, O. (1999, November). An extensional characterization of lambda-lifting and lambda-dropping. In A. Middeldorp and T. Sato (Eds.), *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, Volume 1722 of *Lecture Notes in Computer Science*, pp. 241–250. Springer-Verlag.

de la Briandais, R. (1959). File searching using variable length keys. In *Proc. Western Joint Computer Conference*, Volume 15, pp. 295–298. AFIPS Press.

de Moor, O. and J. Gibbons (2000, May). Pointwise relational programming. In T. Rus (Ed.), *Proceedings of Algebraic Methodology and Software Technology (AMAST 2000), Iowa*, Volume 1816 of *Lecture Notes in Computer Science*, pp. 371–390. Springer-Verlag.

Gierz, G., K. Hofmann, K. Keimel, J. Lawson, M. Mislove, and D. Scott (1980). *A Compendium of Continuous Lattices*. Springer-Verlag.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph. D. thesis, Université Paris VII.

Gunter, C. and D. Scott (1990). Semantic domains. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Chapter 12, pp. 633–674. Elsevier Science Publishers B.V. (North Holland).

Hagino, T. (1987). *Category Theoretic Approach to Data Types*. Ph. D. thesis, University of Edinburgh.

Hallgren, T. and M. Carlsson (1995, May). Programming with Fudgets. In J. Jeuring and E. Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, Volume 925 of *Lecture Notes in Computer Science*, pp. 137–182. Springer-Verlag.

Harper, R. and G. Morrisett (1995). Compiling polymorphism using intensional type analysis. In ACM (Ed.), *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'95, San Francisco, California*, pp. 130–141. ACM-Press.

Henglein, F. (1993, April). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems 15*(2), 253–289.

Hinze, R. (1999, September). A generic programming extension for Haskell. In E. Meijer (Ed.), *Proceedings of the 3rd Haskell Workshop, Paris, France*. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.

Hinze, R. (2000a, May). Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming 10*(3), 305–317.

Hinze, R. (2000b). Generalizing generalized tries. *Journal of Functional Programming*. To appear.

Hinze, R. (2000c). Manufacturing datatypes. *Journal of Functional Programming, Special Issue on Algorithmic Aspects of Functional Programming Languages*. To appear.

Hinze, R. (2000d, July). Memo functions, polytypically! In J. Jeuring (Ed.), *Proceedings of the 2nd Workshop on Generic Programming, Ponte de Lima, Portugal*, pp. 17–32. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-2000-19.

Hinze, R. (2000e, January). A new approach to generic functional programming. In T. W. Reps (Ed.), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL' 00), Boston, Massachusetts, January 19-21*, pp. 119–132.

Hinze, R. (2000f). Polytypic programming with ease. *Journal of Functional and Logic Programming*. To appear.

Hinze, R. (2000g, July). Polytypic values possess polykinded types. In R. Backhouse and J. Oliveira (Eds.), *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, Volume 1837 of *Lecture Notes in Computer Science*, pp. 2–27. Springer-Verlag.

Hinze, R. and S. Peyton Jones (2000, September). Derivable type classes. In G. Hutton (Ed.), *Proceedings of the 4th Haskell Workshop, Montreal, Canada*. The proceedings will be published as a University of Nottingham technical report.

Hoogendijk, P. and R. Backhouse (1997). When do datatypes commute? In E. Moggi and G. Rosolini (Eds.), *Proceedings of the 7th International Conference on Category Theory and Computer Science (Santa Margherita Ligure, Italy, September 4–6)*, Volume 1290 of *Lecture Notes in Computer Science*, pp. 242–260. Springer-Verlag.

Hoogendijk, P. and O. de Moor (2000). Container types categorically. *Journal of Functional Programming 10*(2), 91–225.

Hudak, P. (2000). *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

Hughes, J. (2000, May). Generalising monads to arrows. *Science of Computer Programmming 37*, 67–111.

Huwig, H. and A. Poigné (1990, June). A note on inconsistencies caused by fixpoints in a Cartesian closed category. *Theoretical Computer Science 73*(1), 101–112.

Jansson, P. and J. Jeuring (1997, January). PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, pp. 470–482. ACM-Press.

Jansson, P. and J. Jeuring (1998, June). PolyLib—A library of polytypic functions. In R. Backhouse and T. Sheard (Eds.), *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden*. Department of Computing Science, Chalmers University of Technology and Göteborg University.

Jansson, P. and J. Jeuring (1999). Polytypic compact printing and parsing. In S. D. Swierstra (Ed.), *Proceedings European Symposium on Programming,* *ESOP'99*, Volume 1576 of *Lecture Notes in Computer Science*, Berlin, pp. 273–287. Springer-Verlag.

Jansson, P. and J. Jeuring (2000). Calculating polytypic data conversion programs. *Science of Computer Programmming*. To appear.

Jay, C., G. Bellè, and E. Moggi (1998, November). Functorial ML. *Journal of Functional Programming 8*(6), 573–619.

Jay, C. and J. Cocket (1994, 11–13 April). Shapely types and shape polymorphism. In D. Sanella (Ed.), *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, Volume 788 of *Lecture Notes in Computer Science*, Berlin, pp. 302–316. Springer-Verlag.

Jeuring, J. and P. Jansson (1996). Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard (Eds.), *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, Volume 1129 of *Lecture Notes in Computer Science*, pp. 68–114. Springer-Verlag.

Jones, M. and J. Peterson (1999, May). *Hugs 98 User Manual*. Available from `http://www.haskell.org/hugs`.

Jones, M. P. (1995, January). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming 5*(1), 1–35.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley Publishing Company.

Launchbury, J. and R. Paterson (1996, April). Parametricity and unboxing with unpointed types. In H. R. Nielson (Ed.), *European Symposium on Programming, Linköping, Sweden*, Volume 1058 of *Lecture Notes in Computer Science*, pp. 204–218. Springer-Verlag.

MacLane, S. (1998). *Categories for the Working Mathematician* (2nd ed.). Graduate Texts in Mathematics. Berlin: Springer-Verlag.

McCracken, N. J. (1984). The typechecking of programs with implicit type structure. In G. Kahn, D. B. MacQueen, and G. D. Plotkin (Eds.), *Semantics of Data Types: International Symposium, Sophia-Antipolis, France*, Volume 173 of *Lecture Notes in Computer Science*, pp. 301–315. Springer-Verlag.

Meertens, L. (1996, September). Calculate polytypically! In H. Kuchen and S. Swierstra (Eds.), *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, Volume 1140 of *Lecture Notes in Computer Science*, pp. 1–16. Springer-Verlag.

Meijer, E. and G. Hutton (1995, June). Bananas in space: Extending fold and unfold to exponential types. In *Conference Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 International Conference on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA*, pp. 324–333. ACM-Press.

Michie, D. (1968, April). "Memo" functions and machine learning. *Nature* (218), 19–22.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17*(3), 348–375.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. Cambridge, MA: The MIT Press.

Moggi, E. (1990). An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, Edinburgh University.

Moggi, E. (1991). Notions of computation and monads. *Information and Computation 93*(1), 55–92.

Mycroft, A. (1984). Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet (Eds.), *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, Volume 167 of *Lecture Notes in Computer Science*, pp. 217–228.

O'Donnell, M. J. (1985). *Equational Logic as a Programming Language*. Foundations of Computing Series. Cambridge, Mass.: MIT Press.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press.

Okasaki, C. (1999, September). From fast exponentiation to square matrices: An adventure in types. In P. Lee (Ed.), *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, Paris, France*, pp. 28–35.

Okasaki, C. and A. Gill (1998). Fast mergeable integer maps. In *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, pp. 77–86.

Peyton Jones, S. and J. Hughes (Eds.) (1999, February). *Haskell 98 — A Non-strict, Purely Functional Language*. Available from `http://www.haskell.org/definition/`.

Peyton Jones, S. L. (1996). Compiling Haskell by program transformation: A report from the trenches. In H. R. Nielson (Ed.), *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, 22–24 April*, Volume 1058 of *Lecture Notes in Computer Science*, pp. 18–44. Springer-Verlag.

Plotkin, G. (1983). Domains — Pisa notes on domain theory. The notes have been prepared by Yugo Kashigawa and Hidetaka Kondoh.

Poigné, A. (1992). Basic category theory. In S. Abramsky, D. M. Gabbay, and T. Maibaum (Eds.), *Handbook of Logic in Computer Science, Volume 1, Background: Mathematical Structures*, pp. 413–640. Clarendon Press, Oxford.

Rabhi, F. and G. Lapalme (1999). *Algorithms: a Functional Programming Approach* (second ed.). Addison-Wesley Publishing Company.

Reynolds, J. (1974). Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Paris*, Volume 19 of *Lecture Notes in Computer Science*, pp. 408–425. Springer-Verlag.

Ruehr, F. (1998, June). Structural polymorphism. In R. Backhouse and T. Sheard (Eds.), *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ.

Ruehr, K. F. (1992). *Analytical and Structural Polymorphism Expressed using Patterns over Types*. Ph. D. thesis, University of Michigan.

Sénizergues, G. (1997). The equivalence problem for deterministic pushdown automata is decidable. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela (Eds.), *Automata, Languages and Programming, 24th International Colloquium*, Volume 1256 of *Lecture Notes in Computer Science*, Bologna, Italy, pp. 671–681. Springer-Verlag.

Sheard, T. (1991, October). Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems 13*(4), 531–557.

Sheard, T. (1993, November). Type parametric programming. Technical Report CS/E 93-018, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, Portland, OR, USA.

Taylor, P. (1999). *Practical Foundations of Mathematics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press.

Team, T. G. (2000). *The Glasgow Haskell Compiler User's Guide, Version 4.08*. Available from `http://www.haskell.org/ghc/documentation.html`.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming* (second ed.). Harlow, England: Addison Wesley Longman Limited.

Thue, A. (1912). Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse 1*, 1–67. Reprinted in Thue's "Selected Mathematical Papers" (Oslo: Universitetsforlaget, 1977), 413–477.

Turner, D. (1981, October). The Semantic Elegance of Applicative Languages. In *Functional Programming Languages and Computer Architecture (FPCA '81), Portsmouth, New Hampshire*, pp. 85–92. ACM, New York.

Wadler, P. (1989, September). Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK*, pp. 347–359. Addison-Wesley Publishing Company.

Wadler, P. (1990, June). Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pp. 61–78. ACM-Press.

Wadler, P. (1992, January). The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Sante Fe, New Mexico*, pp. 1–14.

Wadler, P. (1995, May). Monads for functional programming. In J. Jeuring and E. Meijer (Eds.), *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, Volume 925 of *Lecture Notes in Computer Science*, pp. 24–52. Springer-Verlag.

Wadsworth, C. (1979). Recursive type operators which are more than type schemes. *Bulletin of the EATCS 8*, 87–88. Abstract of a talk given at the 2nd International Workshop on the Semantics of Programming Languages, Bad Honnef, Germany, 19–23 March 1979.

Summary

A generic program is one that the programmer writes once, but which works over many different data types. A generic proof is one that the programmer shows once, but which holds for many different data types. This thesis describes a novel approach to functional generic programming and reasoning that is both simpler and more general than previous approaches.

Examples of generic functions are parsing, pretty printing, taking equality, mapping functions, reductions, and so on. We introduce two forms of generic definitions. Definitions of the first form are restricted to type indices of one fixed kind and proceed by induction on the structure of types. Definitions of the second form are more general as they allow the programmer to define values that are indexed by types of arbitrary kinds. It turns out that these type-indexed values possess kind-indexed types, that is, types that are defined by induction on the structure of kinds. Interestingly, to define a kind-indexed type it suffices to specify the image of the base kind; likewise, to define a type-indexed value it suffices to specify the images of type constants. The remaining cases are taken care of automatically, which is one of the strengths of generic programming.

The key idea of our approach is model types by terms of the simply typed lambda calculus augmented by a family of fixed point combinators. The specialization of a generic value can be seen as an interpretation of the simply typed lambda calculus.

For each of the two forms of generic definitions we provide a corresponding proof principle. The first method is a variant of fixed point induction. It can also be used constructively to derive a generic program from its specification. The second method is based on logical relations, one of the main tools for studying typed lambda calculi. To prove a generic property it suffices to prove the assertion for type constants. Again, everything else is taken care of automatically.

We present a multitude of examples of generic values and associated generic proofs. Among other things, we apply the framework to implement dictionaries and memo tables in a generic way. These case studies are particularly interesting in that they make essential use of type-indexed types, that is, types that are defined by induction on the structure of types.

Finally, we show how to extend the functional programming language Haskell 98 by generic definitions. The implementation of this extension is discussed in detail.

Curriculum Vitæ

Ralf Thomas Walter Hinze
Eudenbergerstraße 13
D-53639 Königswinter

geboren am 2. Juli 1965 in Marl (Westfalen),
Familienstand: ledig, zwei Kinder.

1971 – 1975	August-Döhr-Grundschule in Marl
1975 – 1984	Albert-Schweitzer-Gymnasium in Marl
Apr. 1984	Abschluß mit dem Zeugnis der allgemeinen Hochschulreife
Okt. 1984–Apr. 1990	Studium der Informatik an der Universität Dortmund
Apr. 1990	Abschluß des Studiums als Diplom-Informatiker
Apr. 1990–Sep. 1990	Wissenschaftlicher Angestellter an der Abteilung Informatik der Universität Dortmund
Okt. 1990–Jan. 1996	Wissenschaftlicher Angestellter an der Abteilung Informatik der Universität Bonn
Nov. 1995	Abschluß der Promotion (Dr. rer. nat.) in Informatik an der Mathematisch-Naturwissenschaftlichen Fakultät der Universität Bonn
Feb. 1996–Sep. 1996	Wissenschaftlicher Assistent (C1) an der Abteilung Informatik der Universität Bonn
Okt. 1996–Feb. 1997	Wissenschaftlicher Assistent (C1) an der Abteilung Informatik der Technischen Fakultät der Universität Bielefeld
seit März 1997	Wissenschaftlicher Assistent (C1) an der Abteilung Informatik der Universität Bonn