Tutorial

LNCS 2793

Roland Backhouse
Jeremy Gibbons (Eds.)

Generic Programming

Advanced Lectures

Springer

Lecture Notes in Computer Science 2793
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Roland Backhouse Jeremy Gibbons (Eds.)

Generic Programming

Advanced Lectures

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Roland Backhouse
The University of Nottingham
School of Computer Science and Information Technology
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK
E-mail: rcb@cs.nott.ac.uk

Jeremy Gibbons
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
E-mail: Jeremy.Gibbons@comlab.ox.ac.uk

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.3, D.1, D.2, F.3, E.1

ISSN 0302-9743
ISBN 3-540-20194-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10931868 06/3142 5 4 3 2 1 0

Preface

Generic programming is about making programming more effective by making it more general. This volume is about a novel form of genericity in programs, based on parameterizing programs by the structure of the data they manipulate. The material is based on lectures presented at a summer school on Generic Programming held at the University of Oxford in August 2002.

The lectures by Hinze and Jeuring introduced Generic Haskell, an extension of the Haskell programming language that allows the programmer to define a function by induction on the structure of types. The implementation of Generic Haskell provided a valuable tool for students to experiment with applications of this form of datatype genericity. The lecture material in this volume is divided into two parts. The first part ("practice and theory") introduces Generic Haskell and the theory that underlies its design. The second part ("applications") discusses three advanced applications of Generic Haskell in some depth.

The value of generic programming is illusory unless the nature and extent of the genericity can be described clearly and precisely. The lectures by Backhouse and Crole delve deeper into the theoretical basis for datatype genericity. Backhouse reviews the notion of parametric polymorphism (a notion well known to functional programmers) and then shows how this notion is extended to higher-order notions of parametricity. These are used to characterize what it means for a value to be stored in a datatype. Also, transformations on data structures are given precise specifications in this way. Underlying this account are certain basic notions of category theory and allegory theory. Crole presents the category theory needed for a deeper understanding of mechanisms for defining datatypes.

The final chapter, by Fiadeiro, Lopes and Wermelinger applies the mathematical "technology" of parameterization to the larger-scale architectural structure of programs. The description of a system is split into components and their interactions; architectural connectors are parameterized by components, leading to an overall system structure consisting of components and connector instances establishing the interactions between the components.

Our thanks go to all those involved in making the school a success. We are grateful to the technical support staff of the Oxford University Computing Laboratory for providing computing facilities, to Yorck Hunke, David Lacey and Silvija Seres of OUCL for assistance during the school, and to St. Anne's College for an amenable environment for study. Thanks also go to Peter Buneman and Martin Odersky, who lectured at the school on *Semi-structured Data* and on *Object-Oriented and Functional Approaches to Compositional Programming*, respectively, but were unable to contribute to the proceedings.

June, 2003

Roland Backhouse
Jeremy Gibbons

Contributors

Roland Backhouse
School of Computer Science and Information Technology,
University of Nottingham, Nottingham, NG8 1BB, UK
rcb@cs.nott.ac.uk
http://www.cs.nott.ac.uk/~rcb/

Roy Crole
Department of Mathematics and Computer Science,
University of Leicester, University Road, Leicester, LE1 7RH, UK
roy.crole@mcs.le.ac.uk
http://www.mcs.le.ac.uk/~rcrole/

José Luiz Fiadeiro
Department of Computer Science, University of Leicester, University Road,
Leicester, LE1 7RH, UK
jose@fiadeiro.org

Ralf Hinze
Institut für Informatik III, Universität Bonn, Römerstraße 164,
53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

Paul Hoogendijk
Philips Research, Prof. Holstlaan 4, 5655 AA Eindhoven, The Netherlands
Paul.Hoogendijk@philips.com

Johan Jeuring
Institute of Information and Computing Sciences, Utrecht University, P.O.
Box 80.089, 3508 TB Utrecht, The Netherlands
and
Open University, Heerlen, The Netherlands
johanj@cs.uu.nl
http://www.cs.uu.nl/~johanj/

Antónia Lopes
Department of Informatics, Faculty of Sciences, University of Lisbon,
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt

Michel Wermelinger
Dept. of Informatics, Faculty of Sciences and Technology, New University
of Lisbon, Quinta da Torre, 2829-516 Caparica, Portugal
mw@di.fct.unl.pt

Table of Contents

Generic Haskell: Practice and Theory

Ralf Hinze[1] and Johan Jeuring[2,3]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de
http://www.informatik.uni-bonn.de/~ralf/

[2] Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
johanj@cs.uu.nl
http://www.cs.uu.nl/~johanj/

[3] Open University, Heerlen, The Netherlands

Abstract. Generic Haskell is an extension of Haskell that supports the construction of generic programs. These lecture notes describe the basic constructs of Generic Haskell and highlight the underlying theory.

Generic programming aims at making programming more effective by making it more general. Generic programs often embody non-traditional kinds of polymorphism. Generic Haskell is an extension of Haskell [38] that supports the construction of generic programs. Generic Haskell adds to Haskell the notion of *structural polymorphism*, the ability to define a function (or a type) by induction on the structure of types. Such a function is generic in the sense that it works not only for a specific type but for a whole class of types. Typical examples include equality, parsing and pretty printing, serialising, ordering, hashing, and so on.

The lecture notes on Generic Haskell are organized into two parts. This first part motivates the need for genericity, describes the basic constructs of Generic Haskell, puts Generic Haskell into perspective, and highlights the underlying theory. The second part entitled "Generic Haskell: applications" delves deeper into the language discussing three non-trivial applications of Generic Haskell: generic dictionaries, compressing XML documents, and a generic version of the zipper data type.

The first part is organized as follows. Section 1 provides some background discussing type systems in general and the type system of Haskell in particular. Furthermore, it motivates the basic constructs of Generic Haskell. Section 2 takes a closer look at generic definitions and shows how to define some popular generic functions. Section 3 highlights the theory underlying Generic Haskell and discusses its implementation. Section 4 concludes.

1 Introduction

This section motivates and introduces the basic constructs of Generic Haskell. We start by looking at type systems.

A basic knowledge of *Haskell* is desirable, as all the examples are given either in Haskell or in Generic Haskell.

1.1 Type Systems

Safe Languages. Most programmers probably agree that *language safety* is a good thing. Language safety is quite a colorful term meaning different things to different people. Here are a few definitions taken from Pierce's excellent text book "Types and Programming Languages" [40].

– A *safe language* is one that makes it impossible to shoot yourself in the foot while programming.
– A *safe language* is one that protects its own abstractions.
– A *safe language* is one that that prevents untrapped errors at run time.
– A *safe language* is completely defined by its programmer's manual.

The definitions put emphasis on different aspects of language safety. Quite clearly, all of these are desirable properties of a programming language.

Now, language safety can be achieved by *static type checking*, by *dynamic type checking*, or—and this is the most common case—by a combination of static and dynamic checks. The language Haskell serves as an example of the latter approach: passing an integer to a list-processing function is captured statically at compile time while taking the first element of the empty list results in a run-time error.

Static and Dynamic Typing. It is widely accepted that static type systems are indispensable for building large and reliable software systems. The most cited benefits of static typing include:

– Programming errors are detected at an early stage.
– Type systems enforce disciplined programming.
– Types promote abstraction (abstract data types, module systems).
– Types provide machine-checkable documentation.

However, type systems are always *conservative*: they must necessarily reject programs that behave well at run time.

In a sense, generic programming is about extending the boundaries of static type systems. This is the reason why these lecture notes have little to offer for addicts of dynamically typed languages. As we will see, most generic programs can be readily implemented in a dynamically checked language. (Conceptually, a dynamic language offers one universal data type; programming a function that works for a class of data types is consequently a non-issue.)

Polymorphic Type Systems. Polymorphism complements type security by flexibility. Polymorphic type systems like the Hindley-Milner system [33] allow the definition of functions that behave uniformly over all types. A standard example is the *length* function that computes the length of a list.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{List\ a} && = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a}\ (\mathsf{List\ a}) \\
&\mathit{length} && :: \forall \mathsf{a}\,.\,\mathsf{List\ a} \to \mathsf{Int} \\
&\mathit{length\ Nil} && = 0 \\
&\mathit{length}\ (\mathit{Cons\ a\ as}) && = 1 + \mathit{length\ as}
\end{aligned}
$$

The first line declares the list data type, which is parametric in the type of list elements. The function *length* happens to be insensitive to the element type. This is signalled by the universal quantifier in *length*'s type signature (read: $\mathsf{List\ a} \to \mathsf{Int}$ is a valid type of *length* for all types a). Though this is not Haskell 98 syntax, we will write polymorphic types always using explicit qualifiers. Most readers probably know the universal quantifier from predicate logic. Indeed, there is a close correspondence between polymorphic type systems and systems of higher-order logic, see [47]. In light of this correspondence we note that the quantifier in *length*'s type signature is *second-order* as it ranges over sets (if we naively equate types with sets).

However, even polymorphic type systems are sometimes less flexible than one would wish. For instance, it is not possible to define a polymorphic *equality function* that works for all types.

$$\mathit{eq} :: \forall \mathsf{a}\,.\,\mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- does not work}$$

The parametricity theorem [46] implies that a function of type $\forall \mathsf{a}\,.\,\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ must necessarily be constant. As a consequence, the programmer is forced to program a separate equality function for each type from scratch. This sounds like a simple task but may, in fact, be arbitrarily involved. To illustrate some of the difficulties we will go through a series of instances of equality (and other generic functions). First, however, let us take a closer look at Haskell's type system, especially at the **data** construct.

1.2 Haskell's data Construct

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

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}.$$

This definition simultaneously introduces a new type constructor B and n data or value constructors $K_1, \ldots, K_n$, whose types are given by

$$K_j :: \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_m\,.\,\mathsf{t}_{j1} \to \cdots \to \mathsf{t}_{jm_j} \to \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m.$$

The type parameters $\mathsf{a}_1, \ldots, \mathsf{a}_m$ 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, B may also appear on the right-hand side. In general, data types are defined by a system of mutually recursive data type declarations. A Haskell data type is essentially a *sum of products*: the components of the sum are labelled by constructor names; the arguments of a constructor form a product.

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

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

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

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

$$\textbf{data}\ \mathsf{Fork\ a} = \mathit{Fork}\ \mathsf{a\ a}$$

An element of Fork a is a pair whose two components both have type a. This example illustrates that we can use the same name for a type and for a data constructor. In these notes we distinguish the two by using different fonts: data constructors are set in Roman and type constructors in Sans Serif.

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 $\kappa \to \nu$ represents type constructors that map type constructors of kind κ to those of kind ν. Note that the term 'type' is sometimes reserved for nullary type constructors.

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

$$\textbf{data}\ \mathsf{Maybe\ a} = \mathit{Nothing} \mid \mathit{Just}\ \mathsf{a}$$

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

Recursive Types. Data type declarations may be recursive or even mutually recursive. A simple recursive data type is the type of natural numbers.

$$\textbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$$

The number 6, for instance, is given by

$$\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero}))))).$$

Strings can also be represented by a recursive data type.

$$\textbf{data}\ \mathsf{String} = \mathit{NilS} \mid \mathit{ConsS}\ \mathsf{Char\ String}$$

The type String is a binary sum. The first summand, *Nil*, is a nullary product and the second summand, *Cons*, is a binary product. Here is an example element of String:

$$\begin{array}{l} \mathit{ConsS}\ \texttt{'F'}\ (\mathit{ConsS}\ \texttt{'l'}\ (\mathit{ConsS}\ \texttt{'o'}\ (\mathit{ConsS}\ \texttt{'r'} \\ \quad (\mathit{ConsS}\ \texttt{'i'}\ (\mathit{ConsS}\ \texttt{'a'}\ (\mathit{ConsS}\ \texttt{'n'}\ \mathit{NilS})))))). \end{array}$$

The most popular data type is without doubt the type of parametric lists; it is obtained from String by abstracting over Char.

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

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

$$\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 5\ (\mathit{Cons}\ 7\ (\mathit{Cons}\ 11\ (\mathit{Cons}\ 13\ \mathit{Nil}))))).$$

In Haskell, lists are predefined with special syntax: $\mathsf{List}\ \mathsf{a}$ is written $[\mathsf{a}]$, *Nil* is replaced by $[\,]$, and $\mathit{Cons}\ a\ x$ by $a : x$. We will use both notations simultaneously.

The following definition introduces external binary search trees.

$$\textbf{data}\ \mathsf{Tree}\ \mathsf{a}\ \mathsf{b} = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{Tree}\ \mathsf{a}\ \mathsf{b})\ \mathsf{b}\ (\mathsf{Tree}\ \mathsf{a}\ \mathsf{b})$$

We distinguish between external nodes of the form $\mathit{Tip}\ a$ 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 $\mathsf{Tree}\ \mathsf{Bool}\ \mathsf{Int}$:

$$\mathit{Node}\ (\mathit{Tip}\ \mathit{True})\ 7\ (\mathit{Node}\ (\mathit{Tip}\ \mathit{True})\ 9\ (\mathit{Tip}\ \mathit{False})).$$

The type Tree has kind $\star \to \star \to \star$. Perhaps surprisingly, binary type constructors like Tree are curried in Haskell.

The following data type declaration captures multiway branching trees, also known as *rose trees* [6].

$$\textbf{data}\ \mathsf{Rose}\ \mathsf{a} = \mathit{Branch}\ \mathsf{a}\ (\mathsf{List}\ (\mathsf{Rose}\ \mathsf{a}))$$

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

$$\begin{array}{l} \mathit{Branch}\ 2\ (\mathit{Cons}\ (\mathit{Branch}\ 3\ \mathit{Nil}) \\ \qquad\qquad (\mathit{Cons}\ (\mathit{Branch}\ 5\ \mathit{Nil}) \\ \qquad\qquad (\mathit{Cons}\ (\mathit{Branch}\ 7\ (\mathit{Cons}\ (\mathit{Branch}\ 11\ \mathit{Nil}) \\ \qquad\qquad\qquad\qquad\qquad (\mathit{Cons}\ (\mathit{Branch}\ 13\ \mathit{Nil})\ \mathit{Nil})))\ \mathit{Nil}))). \end{array}$$

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

$$\begin{array}{l} \textbf{data}\ \mathsf{Rose}'\ \mathsf{a}\ \ = \mathit{Branch}'\ \mathsf{a}\ (\mathsf{Forest}\ \mathsf{a}) \\ \textbf{data}\ \mathsf{Forest}\ \mathsf{a} = \mathit{NilF} \mid \mathit{ConsF}\ (\mathsf{Rose}'\ \mathsf{a})\ (\mathsf{Forest}\ \mathsf{a}). \end{array}$$

Now Rose' depends on Forest and vice versa.

The type parameters of a data type may range over type constructors of arbitrary kinds. By contrast, Miranda (trademark of Research Software Ltd), Standard ML, and previous versions of Haskell (1.2 and before) only have first-order kinded data types. The following generalization of rose trees, that abstracts over the List data type, illustrates this feature.

$$\textbf{data}\ \mathsf{GRose\ f\ a} = \mathit{GBranch}\ \mathsf{a}\ (\mathsf{f}\ (\mathsf{GRose\ f\ a}))$$

A slight variant of this definition has been used by [37] 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

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

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 [32] where it is employed to give a generic definition of so-called *cata-* and *anamorphisms* [30].

$$\begin{aligned} &\textbf{newtype}\ \mathsf{Fix\ f} &&= \mathit{In}\ (\mathsf{f}\ (\mathsf{Fix\ f})) \\ &\textbf{data}\ \mathsf{ListBase\ a\ b} &&= \mathit{NilL} \mid \mathit{ConsL}\ \mathsf{a\ b} \end{aligned}$$

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

$$\textbf{type}\ \mathsf{List\ a} = \mathsf{Fix}\ (\mathsf{ListBase\ a}).$$

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

$$\begin{aligned} &\mathit{In}\ (\mathit{ConsL}\ 2\ (\mathit{In}\ (\mathit{ConsL}\ 3\ (\mathit{In}\ (\mathit{ConsL}\ 5 \\ &\qquad (\mathit{In}\ (\mathit{ConsL}\ 7\ (\mathit{In}\ (\mathit{ConsL}\ 11\ (\mathit{In}\ (\mathit{ConsL}\ 13\ (\mathit{In}\ \mathit{NilL})))))))\ ))))). \end{aligned}$$

Nested Types. A *regular* or *uniform* data type is a recursive, 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 [7]. Nested data types are practically important since they can capture data-structural invariants in a way that regular data types cannot. For instance, the following data type declaration defines perfectly balanced, binary leaf trees [20]—perfect trees for short.

$$\textbf{data}\ \mathsf{Perfect\ a} = \mathit{ZeroP}\ \mathsf{a} \mid \mathit{SuccP}\ (\mathsf{Perfect}\ (\mathsf{Fork\ a}))$$

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:

```
SuccP (SuccP (SuccP (ZeroP (Fork (Fork (Fork 2 3)
                                        (Fork 5 7))
                                  (Fork (Fork 11 13)
                                        (Fork 17 19)) )))).
```

Note that the height of the perfect tree is encoded in the prefix of *SuccP* and *ZeroP* constructors.

The next data type provides an alternative to the ubiquitous list type if an efficient indexing operation is required: Okasaki's *binary random-access lists* [37] support logarithmic access to the elements of a list.

```
data Sequ a = EndS
            | ZeroS (Sequ (Fork a))
            | OneS a (Sequ (Fork a))
```

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:

```
ZeroS (OneS (Fork 2 3) (OneS (Fork (Fork 5 7) (Fork 11 13)) EndS)).
```

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 [7]:

```
data Bush a = NilB | ConsB a (Bush (Bush a)).
```

An element of type Bush a resembles an ordinary list except that the i-th element has type Bush^i a rather than a. Here is an example element of type Bush Int:

```
ConsB 1 (ConsB (ConsB 2 NilB)
               (ConsB (ConsB (ConsB 3 NilB) NilB) NilB)).
```

Perhaps surprisingly, we will get to know a practical application of this data type in the second part of these notes, which deals with so-called *generalized tries*.

Haskell's **data** construct is surprisingly expressive. In fact, all primitive data types such as characters or integers can, in principle, be defined by a data declaration. The only exceptions to this rule are the function space constructor and Haskell's IO data type. Now, the one-million-dollar question is, of course, how can we define a function that works for all of these data types.

1.3 Towards Generic Programming

The basic idea of generic programming is to define a function such as 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.

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, we 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 these notes. For the moment, let us support the claim by defining two simple algorithms both in a conventional and in a generic style (data compression and equality). Of course, these are 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.

Towards Generic Data Compression. 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 [25]. For concreteness, we assume that bit streams are given by the following data type (we use Haskell's predefined list data type here):

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

Thus, a bit stream is simply a list of bits. A real implementation might have a more sophisticated representation for Bin but that is a separate matter.

We will implement binary encoders and decoders for three different data types. We consider these types in increasing level of difficulty. The first example type is String. Supposing that $\mathit{encodeChar} :: \mathsf{Char} \to \mathsf{Bin}$ is an encoder for characters provided from somewhere, we can encode an element of type String as follows:

$$\begin{array}{ll} \mathit{encodeString} & :: \mathsf{String} \to \mathsf{Bin} \\ \mathit{encodeString}\ \mathit{NilS} & = \mathtt{0} : [\,] \\ \mathit{encodeString}\ (\mathit{ConsS}\ c\ s) & = \mathtt{1} : \mathit{encodeChar}\ c \mathbin{+\!\!+} \mathit{encodeString}\ s. \end{array}$$

We emit one bit to distinguish between the two constructors *NilS* and *ConsS*. If the argument is a non-empty string of the form $\mathit{ConsS}\ c\ s$, 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.

$$\begin{array}{ll} \mathit{decodesString} & :: \mathsf{Bin} \to (\mathsf{String}, \mathsf{Bin}) \\ \mathit{decodesString}\ [\,] & = \mathit{error}\ \texttt{"decodesString"} \end{array}$$

$$\begin{array}{ll} \mathit{decodesString}\ (\mathtt{0} : \mathit{bin}) & = (\mathit{NilS}, \mathit{bin}) \\ \mathit{decodesString}\ (\mathtt{1} : \mathit{bin}) & = \textbf{let}\ (c, \mathit{bin}_1) = \mathit{decodesChar}\ \mathit{bin} \\ & \qquad\quad\ (s, \mathit{bin}_2) = \mathit{decodesString}\ \mathit{bin}_1 \\ & \quad\ \textbf{in}\ (\mathit{ConsS}\ c\ s, \mathit{bin}_2) \end{array}$$

The decoder has type $\mathsf{Bin} \to (\mathsf{String}, \mathsf{Bin})$ rather than $\mathsf{Bin} \to \mathsf{String}$ to be able to compose decoders in a modular fashion: *decodesChar* :: $\mathsf{Bin} \to (\mathsf{Char}, \mathsf{Bin})$, 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).

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

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

A string is a list of characters. We have seen that we obtain Haskell's list type by abstracting over the type of list elements. 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:

$$\begin{array}{ll}
\mathit{encodeListBin} & :: \mathsf{List\ Bin} \to \mathsf{Bin}\\
\mathit{encodeListBin}\ \mathit{Nil} & = \mathsf{0} : [\,]\\
\mathit{encodeListBin}\ (\mathit{Cons}\ \mathit{bin}\ \mathit{bins}) & = \mathsf{1} : \mathit{bin} + \!\!\! + \ \mathit{encodeListBin}\ \mathit{bins}.
\end{array}$$

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

$$\begin{array}{ll}
\mathit{mapList} & :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List\ a}_1 \to \mathsf{List\ a}_2)\\
\mathit{mapList}\ \mathit{mapa}\ \mathit{Nil} & = \mathit{Nil}\\
\mathit{mapList}\ \mathit{mapa}\ (\mathit{Cons}\ a\ \mathit{as}) & = \mathit{Cons}\ (\mathit{mapa}\ a)\ (\mathit{mapList}\ \mathit{mapa}\ \mathit{as}).
\end{array}$$

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 these notes). Combining *encodeListBin* and *mapList* we can encode a variety of lists:

$$\begin{array}{l}
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeChar}\\
\qquad (\mathit{Cons}\ \texttt{'A'}\ (\mathit{Cons}\ \texttt{'n'}\ (\mathit{Cons}\ \texttt{'j'}\ (\mathit{Cons}\ \texttt{'a'}\ \mathit{Nil}))))\,)\\
\Longrightarrow \mathsf{1100000101011101101010101101100001100}\\
\mathit{encodeListBin}\ (\mathit{mapList}\ \mathit{encodeInt}\ (\mathit{Cons}\ 47\ (\mathit{Cons}\ 11\ \mathit{Nil})))\\
\Longrightarrow \mathsf{11111010000000000111010000000000000}\\
(\mathit{encodeListBin} \cdot \mathit{mapList}\ (\mathit{encodeListBin} \cdot \mathit{mapList}\ \mathit{encodeBool}))\\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ \mathit{Nil})))\\
\qquad (\mathit{Cons}\ (\mathit{Cons}\ \mathit{False}\ (\mathit{Cons}\ \mathit{True}\ (\mathit{Cons}\ \mathit{False}\ \mathit{Nil})))\\
\qquad (\mathit{Nil})))\\
\Longrightarrow \mathsf{11110110110111000}.
\end{array}$$

Here, *encodeInt* and *encodeBool* are primitive encoders for integers and Boolean values respectively (an integer occupies 16 bits whereas a Boolean value makes do with one bit).

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.

```
decodesList                :: ∀a . (Bin → (a, Bin)) → (Bin → (List a, Bin))
decodesList dea []         = error "decodesList"
decodesList dea (0 : bin)  = (Nil, bin)
decodesList dea (1 : bin)  = let (a, bin₁) = dea bin
                                 (as, bin₂) = decodesList dea bin₁
                             in (Cons a as, bin₂)
```

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

```
encodeList                 :: ∀a . (a → Bin) → (List a → Bin)
encodeList ena Nil         = 0 : []
encodeList ena (Cons a as) = 1 : ena a ++ encodeList ena as.
```

It is not hard to see that *encodeList ena* = *encodeListBin* · *mapList ena*. Encoding and decoding lists is now fairly simple:

```
encodeList encodeChar (Cons 'A' (Cons 'n' (Cons 'j' (Cons 'a' Nil))))
⟹ 1100000101011101101010101101100001100
encodeList encodeInt (Cons 47 (Cons 11 Nil))
⟹ 11111010000000000111010000000000000
encodeList (encodeList encodeBool)
    (Cons (Cons True (Cons False (Cons True Nil)))
    (Cons (Cons False (Cons True (Cons False Nil)))
    (Nil)))
⟹ 11110110110111000.
```

The third data type we look at is Okasaki's *binary random-access list*. Using the recursion scheme of *encodeList* we can also program an encoder for binary random-access lists.

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

encodeSequ                  :: ∀a . (a → Bin) → (Sequ a → Bin)
encodeSequ ena EndS         = 0 : []
encodeSequ ena (ZeroS s)    = 1 : 0 : encodeSequ (encodeFork ena) s
encodeSequ ena (OneS a s)   = 1 : 1 : ena a ++ encodeSequ (encodeFork ena) s
```

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 *ena*. Similarly, since s has type Sequ (Fork a), we call *encodeSequ* (*encodeFork ena*). It is not hard to see that the type of the component determines the function calls in a straightforward manner. As an aside, note that *encodeSequ* requires a non-schematic form of recursion known as *polymorphic recursion* [36]. The two recursive calls are at type (Fork a → Bin) → (Sequ (Fork a) → 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 [17].

Though the Sequ data type is more complex than the list data type, encoding binary random-access lists is not any more difficult.

```
encodeSequ encodeChar (ZeroS (ZeroS (OneS
     (Fork (Fork 'L' 'i') (Fork 's' 'a')) EndS)))
==> 101011001100101001011011001110100001100
encodeSequ encodeInt (ZeroS (OneS (Fork 47 11) EndS))
==> 1011111101000000000011010000000000000
```

In general, a string of length n makes do with $2 * \lceil lg\ (n + 1)\rceil + 1 + 8 * n$ bits. Perhaps surprisingly, encoding a binary random-access list requires fewer bits than encoding the corresponding list (if the list contains more than 8 elements).

To complete the picture here is the decoder for binary random-access lists.

```
decodesFork                :: ∀a . (Bin → (a, Bin)) → (Bin → (Fork a, Bin))
decodesFork dea bin        = let (a₁, bin₁) = dea bin
                                 (a₂, bin₂) = dea bin₁
                             in (Fork a₁ a₂, bin₂)
decodesSequ                :: ∀a . (Bin → (a, Bin)) → (Bin → (Sequ a, Bin))
decodesSequ dea []         = error "decodes"
decodesSequ dea (0 : bin)  = (EndS, bin)
decodesSequ dea (1 : 0 : bin)
                           = let (s, bin′) = decodesSequ (decodesFork dea) bin
                             in (ZeroS s, bin′)
decodesSequ dea (1 : 1 : bin)
                           = let (a, bin₁) = dea bin
                                 (s, bin₂) = decodesSequ (decodesFork dea) bin₁
                             in (OneS a s, bin₂)
```

Towards Generic Equality. As a second example, let us work towards implementing a generic version of equality. Taking a look at several ad-hoc instances of equality will improve our understanding when we consider the generic programming extensions Generic Haskell offers.

Let us start simple: here is equality of strings.

$$\begin{array}{ll}
\mathit{eqString} & :: \mathsf{String} \to \mathsf{String} \to \mathsf{Bool} \\
\mathit{eqString}\ \mathit{NilS}\ \mathit{NilS} & = \mathit{True} \\
\mathit{eqString}\ \mathit{NilS}\ (\mathit{ConsS}\ c'\ s') & = \mathit{False} \\
\mathit{eqString}\ (\mathit{ConsS}\ c\ s)\ \mathit{NilS} & = \mathit{False} \\
\mathit{eqString}\ (\mathit{ConsS}\ c\ s)\ (\mathit{ConsS}\ c'\ s') & = \mathit{eqChar}\ c\ c' \wedge \mathit{eqString}\ s\ s'
\end{array}$$

The function $\mathit{eqChar} :: \mathsf{Char} \to \mathsf{Char} \to \mathsf{Bool}$ is equality of characters. As usual, we assume that this function is predefined.

The type List is obtained from String by abstracting over Char. Likewise, *eqList* is obtained from *eqString* by abstracting over *eqChar*.

$$\begin{array}{ll}
\mathit{eqList} & :: \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \\
 & \qquad \to (\mathsf{List}\ \mathsf{a} \to \mathsf{List}\ \mathsf{a} \to \mathsf{Bool}) \\
\mathit{eqList}\ \mathit{eqa}\ \mathit{Nil}\ \mathit{Nil} & = \mathit{True} \\
\mathit{eqList}\ \mathit{eqa}\ \mathit{Nil}\ (\mathit{Cons}\ a'\ x') & = \mathit{False} \\
\mathit{eqList}\ \mathit{eqa}\ (\mathit{Cons}\ a\ x)\ \mathit{Nil} & = \mathit{False} \\
\mathit{eqList}\ \mathit{eqa}\ (\mathit{Cons}\ a\ x)\ (\mathit{Cons}\ a'\ x') & = \mathit{eqa}\ a\ a' \wedge \mathit{eqList}\ \mathit{eqa}\ x\ x'
\end{array}$$

Similarly, the type GRose of generalized rose trees is obtained from Rose by abstracting over the list type constructor (which is of kind $\star \to \star$). Likewise, *eqGRose* abstracts over list equality (which has a polymorphic type). Thus, *eqGRose* takes a polymorphic function to a polymorphic function.

$$\begin{array}{l}
\mathit{eqGRose} :: \forall \mathsf{f} \,.\, (\forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{a} \to \mathsf{Bool})) \\
\qquad \to (\forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \\
\qquad\qquad \to (\mathsf{GRose}\ \mathsf{f}\ \mathsf{a} \to \mathsf{GRose}\ \mathsf{f}\ \mathsf{a} \to \mathsf{Bool})) \\
\mathit{eqGRose}\ \mathit{eqf}\ \mathit{eqa}\ (\mathit{GBranch}\ a\ f)\ (\mathit{GBranch}\ a'\ f') \\
\quad = \mathit{eqa}\ a\ a' \wedge \mathit{eqf}\ (\mathit{eqGRose}\ \mathit{eqf}\ \mathit{eqa})\ f\ f'
\end{array}$$

The function *eqGRose* has a so-called rank-2 type. In general, the rank of a type is given by

$$\begin{array}{ll}
\mathit{rank}(\mathsf{C}) & = 0 \\
\mathit{rank}(\forall \mathsf{a} \,.\, \mathsf{t}) & = \mathit{max}\{1, \mathit{rank}(\mathsf{t})\} \\
\mathit{rank}(\mathsf{t} \to \mathsf{u}) & = \mathit{max}\{\mathit{inc}\ (\mathit{rank}(\mathsf{t})), \mathit{rank}(\mathsf{u})\},
\end{array}$$

where $\mathit{inc}\ 0 = 0$ and $\mathit{inc}\ (n{+}1) = n{+}2$. Most implementations of Haskell support rank-2 types. The latest version of the Glasgow Haskell Compiler, GHC 5.04, even supports general rank-n types.

As a final example, consider defining equality for the fixed point operator on the type level.

$$\begin{array}{l}
\mathit{eqFix} :: \forall \mathsf{f} \,.\, (\forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{a} \to \mathsf{Bool})) \\
\qquad \to (\mathsf{Fix}\ \mathsf{f} \to \mathsf{Fix}\ \mathsf{f} \to \mathsf{Bool}) \\
\mathit{eqFix}\ \mathit{eqf}\ (\mathit{In}\ f)\ (\mathit{In}\ f') = \mathit{eqf}\ (\mathit{eqFix}\ \mathit{eqf})\ f\ f'
\end{array}$$

1.4 Towards Generic Haskell

In the previous section we have seen a bunch of ad-hoc instances of generic functions. Looking at the type signatures of equality we see that the type of *eqT* depends on the kind of T. Roughly speaking, the more complicated the kind of T, the more complicated the type of *eqT*. To capture the type of generic functions, Generic Haskell supports the definition of types that are defined by induction over the structure of kinds, so-called *kind-indexed types*.

Apart from the typings, it is crystal clear what the definition of *eqT* looks like. We first have to check whether the two arguments of equality are labelled by the same constructor. If this is the case, then their arguments are recursively tested for equality. Nonetheless, coding the equality function is boring and consequently error-prone. Fortunately, Generic Haskell allows us to capture equality once and for all.

To define generic equality and other generic functions it suffices to cover three simple, non-recursive data types: binary sums, binary products and nullary products (that is, the unit data type). Since these types are the building blocks of data declarations, Generic Haskell is then able to generate instances of equality for arbitrary user-defined types. In other words, the generic equality function works for all types of all kinds. Of course, if the user-defined type falls back on some primitive type, then the generic equality function must also supply code for this type. Thus, generic equality will include cases for Char, Int, etc. On the other hand, it will not include cases for function types or for the IO type since we cannot decide equality of functions or IO actions.

We have already mentioned the slogan that generic programming gives the programmer a lot of things for free. In our case, Generic Haskell automatically takes care of type abstraction, type application and type recursion. And it does so in a type-safe manner.

Kind-Indexed Types. The type of a generic function is captured by a kind-indexed type which is defined by induction on the structure of kinds. Here are some examples.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Encode}\{[\star]\}\ \mathsf{t} && = \mathsf{t} \to \mathsf{Bin}\\
&\textbf{type}\ \mathsf{Encode}\{[\kappa \to \nu]\}\ \mathsf{t} && = \forall \mathsf{a}\,.\,\mathsf{Encode}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Encode}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})\\
&\textbf{type}\ \mathsf{Decodes}\{[\star]\}\ \mathsf{t} && = \mathsf{Bin} \to (\mathsf{t}, \mathsf{Bin})\\
&\textbf{type}\ \mathsf{Decodes}\{[\kappa \to \nu]\}\ \mathsf{t} && = \forall \mathsf{a}\,.\,\mathsf{Decodes}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Decodes}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})\\
&\textbf{type}\ \mathsf{Eq}\{[\star]\}\ \mathsf{t} && = \mathsf{t} \to \mathsf{t} \to \mathsf{Bool}\\
&\textbf{type}\ \mathsf{Eq}\{[\kappa \to \nu]\}\ \mathsf{t} && = \forall \mathsf{a}\,.\,\mathsf{Eq}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Eq}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

The part enclosed in $\{[\cdot]\}$ is the kind index. In each case, the equation for kind $\star$ is the interesting one. For instance, $\mathsf{t} \to \mathsf{t} \to \mathsf{Bool}$ is the type of equality for manifest types (nullary type constructors). Perhaps surprisingly, the equations for function kinds always follow the same scheme, which we will encounter time and again. We will see in Section 3.3 that this scheme is inevitable because of the way type constructors of kind $\kappa \to \nu$ are specialized.

The type signatures we have seen in the previous section can be written more succinctly using the kind-indexed types above.

$$\begin{array}{ll}
\mathit{encodeString} & :: \mathsf{Encode}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{encodeList} & :: \mathsf{Encode}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{decodesString} & :: \mathsf{Decodes}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{decodesList} & :: \mathsf{Decodes}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{eqString} & :: \mathsf{Eq}\{\![\star]\!\}\ \mathsf{String} \\
\mathit{eqList} & :: \mathsf{Eq}\{\![\star \to \star]\!\}\ \mathsf{List} \\
\mathit{eqGRose} & :: \mathsf{Eq}\{\![(\star \to \star) \to (\star \to \star)]\!\}\ \mathsf{GRose} \\
\mathit{eqFix} & :: \mathsf{Eq}\{\![(\star \to \star) \to \star]\!\}\ \mathsf{Fix}
\end{array}$$

In general, the equality function for type t of kind κ has type $\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t}$.

Sums and Products. Recall that a Haskell data type is essentially a sum of products. To cover data types the generic programmer only has to define the generic function for binary sums and binary products (and nullary products). To this end Generic Haskell provides the following data types.

$$\begin{array}{l}
\mathbf{data}\ \mathsf{Unit} \quad = \mathit{Unit} \\
\mathbf{data}\ \mathsf{a} :\!*\!: \mathsf{b} = \mathsf{a} :\!*\!: \mathsf{b} \\
\mathbf{data}\ \mathsf{a} :\!+\!: \mathsf{b} = \mathit{Inl}\ \mathsf{a} \mid \mathit{Inr}\ \mathsf{b}
\end{array}$$

Note that the operator ':∗:' is used both as a type constructor and as a data constructor (pairing).

In a sense, the generic programmer views the data declaration

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

as if it were given by the following type definition

$$\mathbf{type}\ \mathsf{List}\ \mathsf{a} = \mathsf{Unit} :\!+\!: \mathsf{a} :\!*\!: \mathsf{List}\ \mathsf{a},$$

which makes sums and products explicit (':∗:' binds more tightly than ':+:').

The types Unit, ':∗:', and ':+:' are isomorphic to the predefined Haskell types '()', '(,)', and *Either*. The main reason for introducing new types is that it gives the user the ability to provide special instances for '()', '(,)', and *Either*. As an example, you may want to show elements of the pair data type in a special way (by contrast, we have seen above that ':∗:' is used to represent the arguments of a constructor).

Type-Indexed Values. Given these prerequisites, the definition of generic functions is within reach. The generic programmer has to provide a type signature, which typically involves a kind-indexed type, and a set of equations, one for each type constant, where a type constant is either a primitive type like Char, Int, '$\rightarrow$' etc or one of the three types Unit, ':*:', and ':+:'. As an example here is the definition of the generic encoding function.

$$\begin{array}{ll}
\mathit{encode}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Encode}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{encode}\{\!|\mathsf{Char}|\!\} & = \mathit{encodeChar} \\
\mathit{encode}\{\!|\mathsf{Int}|\!\} & = \mathit{encodeInt} \\
\mathit{encode}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} & = [\,] \\
\mathit{encode}\{\!|{:+:}|\!\}\ \mathit{ena}\ \mathit{enb}\ (\mathit{Inl}\ a) & = \mathsf{0} : \mathit{ena}\ a \\
\mathit{encode}\{\!|{:+:}|\!\}\ \mathit{ena}\ \mathit{enb}\ (\mathit{Inr}\ b) & = \mathsf{1} : \mathit{enb}\ b \\
\mathit{encode}\{\!|{:*:}|\!\}\ \mathit{ena}\ \mathit{enb}\ (a :*: b) & = \mathit{ena}\ a +\!\!+\ \mathit{enb}\ b
\end{array}$$

Generic functions are also called *type-indexed values*; the part enclosed in $\{\!|\cdot|\!\}$ is the type index. Note that each equation is more or less inevitable. Characters and integers are encoded using the primitive functions *encodeChar* and *encodeInt*. To encode the single element of the unit type no bits are required. 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. Since the types ':+:' and ':*:' have kind $\star \rightarrow \star \rightarrow \star$, the generic instances take two additional arguments, *ena* and *enb*.

The definition of *decode* follows the same definitional pattern.

$$\begin{array}{ll}
\mathit{decodes}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Decodes}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{decodes}\{\!|\mathsf{Char}|\!\} & = \mathit{decodesChar} \\
\mathit{decodes}\{\!|\mathsf{Int}|\!\} & = \mathit{decodesInt} \\
\mathit{decodes}\{\!|\mathsf{Unit}|\!\}\ \mathit{bin} & = (\mathit{Unit}, \mathit{bin}) \\
\mathit{decodes}\{\!|{:+:}|\!\}\ \mathit{dea}\ \mathit{deb}\ [\,] & = \mathit{error}\ \texttt{"decodes"} \\
\mathit{decodes}\{\!|{:+:}|\!\}\ \mathit{dea}\ \mathit{deb}\ (\mathsf{0} : \mathit{bin}) & = \mathbf{let}\ (a, \mathit{bin}') = \mathit{dea}\ \mathit{bin}\ \mathbf{in}\ (\mathit{Inl}\ a, \mathit{bin}') \\
\mathit{decodes}\{\!|{:+:}|\!\}\ \mathit{dea}\ \mathit{deb}\ (\mathsf{1} : \mathit{bin}) & = \mathbf{let}\ (b, \mathit{bin}') = \mathit{deb}\ \mathit{bin}\ \mathbf{in}\ (\mathit{Inr}\ b, \mathit{bin}') \\
\mathit{decodes}\{\!|{:*:}|\!\}\ \mathit{dea}\ \mathit{deb}\ \mathit{bin} & = \mathbf{let}\ (a, \mathit{bin}_1) = \mathit{dea}\ \mathit{bin} \\
 & \qquad\quad (b, \mathit{bin}_2) = \mathit{deb}\ \mathit{bin}_1 \\
 & \quad\ \mathbf{in}\ ((a :*: b), \mathit{bin}_2)
\end{array}$$

Generic equality is equally straightforward.

$$\begin{array}{ll}
\mathit{eq}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{eq}\{\!|\mathsf{Char}|\!\} & = \mathit{eqChar} \\
\mathit{eq}\{\!|\mathsf{Int}|\!\} & = \mathit{eqInt} \\
\mathit{eq}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{eq}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inl}\ a)\ (\mathit{Inl}\ a') & = \mathit{eqa}\ a\ a' \\
\mathit{eq}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inl}\ a)\ (\mathit{Inr}\ b') & = \mathit{False} \\
\mathit{eq}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inr}\ b)\ (\mathit{Inl}\ a') & = \mathit{False} \\
\mathit{eq}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inr}\ b)\ (\mathit{Inr}\ b') & = \mathit{eqb}\ b\ b' \\
\mathit{eq}\{\!|{:*:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (a :*: b)\ (a' :*: b') & = \mathit{eqa}\ a\ a' \wedge \mathit{eqb}\ b\ b'
\end{array}$$

Generic Application. Given the definitions above we can encode and decode elements of arbitrary user-defined data types. The generic functions are invoked by instantiating the type-index to a specific type, where the type can be any *closed* type expression.

```
encode{|String|} (ConsS 'L' (ConsS 'i' (ConsS 's' (ConsS 'a' NilS))))
==> 1001100101100101101110011101100001100
decodes{|String|} 1001100101100101101110011101100001100
==> (ConsS 'L' (ConsS 'i' (ConsS 's' (ConsS 'a' NilS))), [])
encode{|List Char|} (Cons 'A' (Cons 'n' (Cons 'j' (Cons 'a' Nil))))
==> 1100000101011101101010101101100001100
encode{|List Int|} (Cons 47 (Cons 11 Nil))
==> 11111010000000000111010000000000000
encode{|List (List Bool)|}
    (Cons (Cons True (Cons False (Cons True Nil)))
    (Cons (Cons False (Cons True (Cons False Nil)))
    (Nil)))
==> 11110110110111000.
```

In the examples above we call *encode* and *decodes* always for types of kind $\star$. However, the generic functions are far more flexible: we can call them at any type of any kind. The following session illustrates a more general use (here with generic equality).

```
eq{|List Char|} "hello" "Hello"
==> False
let sim c c' = eqChar (toUpper c) (toUpper c')
eq{|List|} sim "hello" "Hello"
==> True
```

If we instantiate the type index to a type constructor, then we have to pass an 'equality function' for the type argument as an additional parameter. Of course, we can pass any function as long as it meets the typing requirements. In the session above, we pass 'an equality test' that ignores case distinctions. Quite clearly, this gives us an extra degree of flexibility.

Generic Abstraction. Abstraction is at the heart of programming. Generic Haskell also supports a simple form of type abstraction. Common usages of generic functions can be captured using generic abstractions.

```
similar{|t :: ⋆ → ⋆|} :: t Char → t Char → Bool
similar{|t|}          =  eq{|t|} sim
```

Note that *similar* is only applicable to type constructors of kind $\star \to \star$.

1.5 Stocktaking

A generic program is one that the programmer writes once, but which works over many different data types. Broadly speaking, generic programming aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined data types. Examples of generic functions include equality, parsing and pretty printing, serialising, ordering, hashing, and so on. A generic function such as a pretty printer or a parser is written once and for all; 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 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. Consider the structure of the language Haskell. If we ignore the module system, Haskell has the three level structure depicted on the right. The lowest level, that is, the level where 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? We have seen that Haskell allows the programmer to define 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.

kinds

types

values

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, [4]. Dependent types will play little rôle in these notes as generic programming is concerned with the opposite direction, where a lower level depends on the same or a higher level. However, using dependent types we can simulate generic programming, see Section 1.6. 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 $\forall \mathsf{a} . \mathsf{List\ a} \to \mathsf{Int}$. 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
types	defined by induction on the structure of kinds	kind-indexed types
types	defined by induction on the structure of types	type-indexed types
values	defined by induction on the structure of types	type-indexed values

Amazingly, we will encounter examples of all sorts of parameterization in the lecture notes (type-indexed types and kind-indexed kinds are covered in the second part).

1.6 Related Work

This section puts Generic Haskell into a broader perspective discussing its limitations, possible generalizations and variations and alternative approaches to generic programming. To illustrate the underlying ideas we will use generic equality as a running example.

Generic Haskell. We have noted in the beginning of Section 1.1 that it is not possible to define a *polymorphic* equality function that works uniformly for all types.

$$eq :: \forall \mathsf{a} . \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- does not work}$$

Consequently, Generic Haskell treats *eq* as a family of functions indexed by type. Deviating from Generic Haskell's syntax *eq*'s type could be written as follows.

$$eq :: \{\!|\mathsf{a} :: \star|\!\} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$$

A moment's reflection reveals that this is really a *dependent type*: the second and the third argument depend on the first argument, which is a type. Of course, since equality may be indexed by types of arbitrary kinds *eq*'s type signature is slightly more complicated.

$$eq :: \forall \kappa . \{\!|\mathsf{a} :: \kappa|\!\} \to \mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a}$$

The universal quantifier, which ranges over kinds, makes explicit that *eq* works for all kinds.

Though the construct $\{\!|\mathsf{a} :: \kappa|\!\} \to \mathsf{t}$ resembles a dependent type, type-indexed functions are not first-class citizens in Generic Haskell. For example, we cannot define a higher-order generic function that takes type-indexed functions to type-indexed functions. The reason for this restriction is that Generic Haskell implements genericity by translating a type-indexed function into a family of (higher-order) polymorphic functions, see Section 3.3.

Type Classes. Haskell's major innovation is its support for *overloading*, based on type classes. For example, the Haskell Prelude defines the class Eq (slightly simplified):

```
class Eq a where
  eq :: a → a → Bool
```

This *class declaration* defines an overloaded top-level function, called *method*, whose type is

$$eq :: \forall \mathsf{a} \,.\, (\mathsf{Eq}\ \mathsf{a}) \Rightarrow \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

Before we can use *eq* on values of, say Int, we must explain how to take equality over Int values:

```
instance Eq Int where
  eq = eqInt.
```

This *instance declaration* makes Int an element of the type class Eq and says 'the *eq* function at type Int is implemented by *eqInt*'. As a second example consider equality of lists. Two lists are equal if they have the same length and corresponding elements are equal. Hence, we require equality over the element type:

```
instance (Eq a) ⇒ Eq (List a) where
  eq Nil Nil                           = True
  eq Nil (Cons a₂ as₂)                 = False
  eq (Cons a₁ as₁) Nil                 = False
  eq (Cons a₁ as₁) (Cons a₂ as₂)       = eq a₁ a₂ ∧ eq as₁ as₂.
```

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 provided explicitly by the programmer for each newly defined data type. There is, however, one exception to this rule. For a handful of built-in classes Haskell provides special support, the so-called '**deriving**' mechanism. For instance, if you define

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

then Haskell generates the 'obvious' code for equality. What 'obvious' means is specified informally in an Appendix of the language definition [38]. Of course, 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 [23, 1].

Haskell translates type classes and instance declarations into a family of polymorphic functions using the so-called *dictionary passing translation*, which is quite similar to the implementation technique of Generic Haskell.

Intensional Type Analysis. The framework of intensional type analysis [16] was originally developed as a means to improve the implementation of polymorphic functions. It was heavily employed in typed intermediate languages not intended for programmers but for compiler writers. The central idea is to pass types or representation of types at run time, which can be analysed and dispatched upon. Thus, equality has the simple type

$$eq :: \forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a} \to \mathsf{Bool} \qquad \text{-- non-parametric } \forall.$$

As in Generic Haskell equality takes an additional type argument and does a case analysis on this argument (using a **typecase**). The resulting code looks quite similar to our definition of equality. The major difference is that the type argument is interpreted at run time whereas Generic Haskell does the analysis at compile time. On the other hand, type-indexed functions are second-class citizens in Generic Haskell whereas in intensional type analysis they have first-class status. Originally, the framework was restricted to data types of kind $\star$, but recent work [49] has lifted this restriction (the generalization is, in fact, inspired by our work on Generic Haskell).

Type Representations. Typed intermediate languages based on intensional type analysis are expressive but rather complex languages. Perhaps surprisingly, dynamic type dispatch can be simulated in a much more modest setting: we basically require a Hindley-Milner type system augmented with existential types. The central idea is to pass *type representations* instead of types. As a first try, if Rep is the type of type representations, we could assign 'eq' the type $\forall \mathsf{a} \,.\, \mathsf{Rep} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$. This approach, however, does not work. The parametricity theorem [46] implies that a function of this type must necessarily ignore its second and its third argument. The trick is to use a parametric type for type representations:

$$eq :: \forall \mathsf{a} \,.\, \mathsf{Rep}\ \mathsf{a} \to \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}.$$

Here $\mathsf{Rep}\ \mathsf{t}$ is the type representation of t. In [9, 5] it is shown how to define a Rep type in Haskell (augmented with existential types). This approach is, however, restricted to types of one fixed kind.

Dependent Types. We have noted above that the type Generic Haskell assigns to equality resembles a dependent type. Thus, it comes as little surprise that we can simulate Generic Haskell in a dependently typed language [2]. In such a language we can define a simple, non-parametric type Rep of type representations. The correspondence between a type and its representative is established by a function *Type* :: Rep $\to \star$ that maps a representation to its type. The signature of equality is then given by

$$eq :: (\mathsf{a} :: \mathsf{Rep}) \to \mathit{Type}\ \mathsf{a} \to \mathit{Type}\ \mathsf{a} \to \mathsf{Bool}.$$

The code of *eq* is similar to what we have seen before; only the typechecking is more involved as it requires reduction of type expressions.

Historical Notes. The concept of functional generic programming trades under a variety of names: F. Ruehr refers to this concept as *structural polymorphism* [42, 41], T. Sheard calls generic functions *type parametric* [44], C.B. Jay and J.R.B. Cocket use the term *shape polymorphism* [27], R. Harper and G. Morrisett [16] coined the phrase *intensional polymorphism*, and J. Jeuring invented the word *polytypism* [28].

The mainstream of generic programming is based on the initial algebra semantics of datatypes, see, for instance [15], and puts emphasis on general recursion operators like mapping functions and catamorphisms (folds). In [43] several variations of these operators are informally defined and algorithms are given that specialize these functions for given datatypes. The programming language *Charity* [10] automatically provides mapping functions and catamorphisms for each user-defined datatype. Since general recursion is not available, Charity is strongly normalizing. *Functorial ML* [26] 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* [24], a precursor of Generic Haskell, offers a special construct for defining generic functions. The generic definitions are similar to ours (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion (see [19] for a more detailed comparison).

Most approaches are restricted to first-order kinded, regular datatypes (or even subsets of this class). One notable exception is the work of F. Ruehr [42], who presents a higher-order language based on a type system related to ours. Genericity is achieved through the use of type patterns which are interpreted at run-time. By contrast, the implementation technique of Generic Haskell does not require the passing of types or representations of types at run-time.

Exercise 1. Prove that a function of type $\forall \mathsf{a} . \mathsf{a} \to \mathsf{a} \to \mathsf{Bool}$ is necessarily constant.

Exercise 2. Define an *instance* of equality for random-access lists.

Exercise 3. Implement a generic version of Haskell's *compare* function, which determines the precise ordering of two elements. Start by defining an appropriate kind-indexed type and then give equations for each of the type constants Unit, ':+:', and ':*:'.

Exercise 4. Miranda offers a primitive function *force* $:: \forall \mathsf{a} \,.\, \mathsf{a} \to \mathsf{a}$ that *fully* evaluates its argument. Implement a generic version of *force* using Haskell's *seq* function (which is of type $\forall \mathsf{a}\ \mathsf{b} \,.\, \mathsf{a} \to \mathsf{b} \to \mathsf{b}$).

Exercise 5 (Difficult). Define a generic function that memoizes a given function. Its kind-indexed type is given by

$$\begin{aligned}
&\textbf{type}\ \mathsf{Memo}\{[\star]\}\ \mathsf{t} && = \forall \mathsf{v} \,.\, (\mathsf{t} \to \mathsf{v}) \to (\mathsf{t} \to \mathsf{v})\\
&\textbf{type}\ \mathsf{Memo}\{[\kappa \to \nu]\}\ \mathsf{t} && = \forall \mathsf{a} \,.\, \mathsf{Memo}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Memo}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a}).
\end{aligned}$$

Note that $\mathsf{Memo}\{[\star]\}\ \mathsf{t}$ is a polymorphic type. *Hint:* $\mathit{memo}\{|\mathsf{t} :: \star|\}\ f$ should yield a closure that does not dependent on the actual argument of f.

2 Generic Haskell—Practice

In this section we look at generic definitions in more detail explaining the various features of Generic Haskell. In particular, we show how to define mapping functions, reductions, and pretty printers generically.

2.1 Mapping Functions

A *mapping function* for a type constructor F of kind $\star \to \star$ lifts a given function of type $\mathsf{a} \to \mathsf{b}$ to a function of type $\mathsf{F}\ \mathsf{a} \to \mathsf{F}\ \mathsf{b}$. In the sequel we show how to define mapping functions so that they work for *all* types of *all* kinds. Before we tackle the generic definition, let us consider some instances first. As an aside, note that the combination of a type constructor and its mapping function is often referred to as a *functor*.

Here is again the all-time favourite, the mapping function for lists.

$$\begin{aligned}
&\mathit{mapList} && :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2)\\
&\mathit{mapList}\ \mathit{mapa}\ \mathit{Nil} && = \mathit{Nil}\\
&\mathit{mapList}\ \mathit{mapa}\ (\mathit{Cons}\ a\ \mathit{as}) && = \mathit{Cons}\ (\mathit{mapa}\ a)\ (\mathit{mapList}\ \mathit{mapa}\ \mathit{as}).
\end{aligned}$$

The mapping function takes a function and applies it to each element of a given list. It is perhaps unusual to call the argument function *mapa*. The reason for this choice will become clear as we go along. For the moment it suffices to bear in mind that the definition of *mapList* rigidly follows the structure of the data type.

We have seen in Section 1.2 that List can alternatively be defined using an explicit fixed point construction.

$$\textbf{type}\ \mathsf{List}'\ \mathsf{a} = \mathsf{Fix}\ (\mathsf{ListBase}\ \mathsf{a}).$$

How can we define the mapping function for lists thus defined? For a start, we define the mapping function for the base functor.

$$\begin{aligned}
&\mathit{mapListBase} &&:: \forall \mathsf{a}_1\ \mathsf{a}_2 . (\mathsf{a}_1 \to \mathsf{a}_2) \to \forall \mathsf{b}_1\ \mathsf{b}_2 . (\mathsf{b}_1 \to \mathsf{b}_2)\\
& &&\quad \to (\mathsf{ListBase}\ \mathsf{a}_1\ \mathsf{b}_1 \to \mathsf{ListBase}\ \mathsf{a}_2\ \mathsf{b}_2)\\
&\mathit{mapListBase\ mapa\ mapb\ NilL} = \mathit{NilL}\\
&\mathit{mapListBase\ mapa\ mapb}\ (\mathit{ConsL\ a\ b})\\
& &&= \mathit{ConsL}\ (\mathit{mapa\ a})\ (\mathit{mapb\ b})
\end{aligned}$$

Since the base functor has two type arguments, its mapping function takes two functions, *mapa* and *mapb*, and applies them to values of type a_1 and b_1, respectively. Even more interesting is the mapping function for Fix

$$\begin{aligned}
&\mathit{mapFix} &&:: \forall \mathsf{f}_1\ \mathsf{f}_2 . (\forall \mathsf{a}_1\ \mathsf{a}_2 . (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{f}_1\ \mathsf{a}_1 \to \mathsf{f}_2\ \mathsf{a}_2))\\
& &&\quad \to (\mathsf{Fix}\ \mathsf{f}_1 \to \mathsf{Fix}\ \mathsf{f}_2)\\
&\mathit{mapFix\ mapf}\ (\mathit{In\ v}) &&= \mathit{In}\ (\mathit{mapf}\ (\mathit{mapFix\ mapf})\ v),
\end{aligned}$$

which takes a polymorphic function as an argument. In other words, *mapFix* has a rank-2 type. The argument function, *mapf*, has a more general type than one would probably expect: it takes a function of type $\mathsf{a}_1 \to \mathsf{a}_2$ to a function of type $\mathsf{f}_1\ \mathsf{a}_1 \to \mathsf{f}_2\ \mathsf{a}_2$. By contrast, the mapping function for List (which like f has kind $\star \to \star$) takes $\mathsf{a}_1 \to \mathsf{a}_2$ to $\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2$. The definition below demonstrates that the extra generality is vital.

$$\begin{aligned}
&\mathit{mapList}' &&:: \forall \mathsf{a}_1\ \mathsf{a}_2 . (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List}'\ \mathsf{a}_1 \to \mathsf{List}'\ \mathsf{a}_2)\\
&\mathit{mapList}'\ \mathit{mapa} &&= \mathit{mapFix}\ (\mathit{mapListBase\ mapa})
\end{aligned}$$

The argument of *mapFix* has type $\forall \mathsf{b}_1\ \mathsf{b}_2 . (\mathsf{b}_1 \to \mathsf{b}_2) \to (\mathsf{ListBase}\ \mathsf{a}_1\ \mathsf{b}_1 \to \mathsf{ListBase}\ \mathsf{a}_2\ \mathsf{b}_2)$, that is, f_1 is instantiated to $\mathsf{ListBase}\ \mathsf{a}_1$ and f_2 to $\mathsf{ListBase}\ \mathsf{a}_2$.

Now, let us define a generic version of *map*. What is the type of the generic mapping function? As a first attempt, we might define

$$\begin{aligned}
&\mathbf{type}\ \mathsf{Map}\{[\star]\}\ \mathsf{t} &&= \mathsf{t} \to \mathsf{t} \qquad\qquad \text{-- WRONG}\\
&\mathbf{type}\ \mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t} &&= \forall \mathsf{a} . \mathsf{Map}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Map}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a}).
\end{aligned}$$

Alas, we have $\mathsf{Map}\{[\star \to \star]\}\ \mathsf{List} = \forall \mathsf{a} . (\mathsf{a} \to \mathsf{a}) \to (\mathsf{List}\ \mathsf{a} \to \mathsf{List}\ \mathsf{a})$, which is not general enough. The solution is to use a two-argument version of the kind-indexed type Map.

$$\begin{aligned}
&\mathbf{type}\ \mathsf{Map}\{[\star]\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \mathsf{t}_1 \to \mathsf{t}_2\\
&\mathbf{type}\ \mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \forall \mathsf{a}_1\ \mathsf{a}_2 . \mathsf{Map}\{[\kappa]\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Map}\{[\nu]\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)\\
&\mathit{map}\{|\mathsf{t} :: \kappa|\} &&:: \mathsf{Map}\{[\kappa]\}\ \mathsf{t}\ \mathsf{t}
\end{aligned}$$

We obtain $\mathsf{Map}\{[\star \to \star]\}\ \mathsf{List}\ \mathsf{List} = \forall \mathsf{a}_1\ \mathsf{a}_2 . (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{List}\ \mathsf{a}_1 \to \mathsf{List}\ \mathsf{a}_2)$ as desired. In the base case $\mathsf{Map}\{[\star]\}\ \mathsf{t}_1\ \mathsf{t}_2$ equals the type of a conversion function. The inductive case has a very characteristic form, which we have already encountered several times. It specifies that a ‘conversion function’ between the type constructors t_1 and t_2 is a function that maps a conversion function between a_1 and a_2 to a conversion function between $\mathsf{t}_1\ \mathsf{a}_1$ and $\mathsf{t}_2\ \mathsf{a}_2$, for all possible instances of a_1 and a_2. Roughly speaking, $\mathsf{Map}\{[\kappa \to \nu]\}\ \mathsf{t}_1\ \mathsf{t}_2$ is the type of a 'conversion function'-transformer. It is not hard to see that the type signatures of *mapList*, *mapListBase*, and *mapFix* are instances of this scheme. Furthermore, from the inductive definition above we can easily conclude that the rank of the type signature corresponds to the kind of the type index: for instance, the *map* for a second-order kinded type has a rank-2 type signature.

The definition of *map* itself is straightforward.

$$
\begin{aligned}
&\mathit{map}\{\!|\mathsf{t} :: \kappa|\!\} && :: \mathsf{Map}\{[\kappa]\}\ \mathsf{t}\ \mathsf{t}\\
&\mathit{map}\{\!|\mathsf{Char}|\!\}\ c && = c\\
&\mathit{map}\{\!|\mathsf{Int}|\!\}\ i && = i\\
&\mathit{map}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} && = \mathit{Unit}\\
&\mathit{map}\{\!|{:}{+}{:}|\!\}\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Inl}\ a) && = \mathit{Inl}\ (\mathit{mapa}\ a)\\
&\mathit{map}\{\!|{:}{+}{:}|\!\}\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Inr}\ b) && = \mathit{Inr}\ (\mathit{mapb}\ b)\\
&\mathit{map}\{\!|{:}{*}{:}|\!\}\ \mathit{mapa}\ \mathit{mapb}\ (a\ {:}{*}{:}\ b) && = \mathit{mapa}\ a\ {:}{*}{:}\ \mathit{mapb}\ b
\end{aligned}
$$

This definition contains all the ingredients needed to derive *maps* for arbitrary data types of arbitrary kinds. As an aside, note that we can define *map* even more succinctly if we use a point-free style—as usual, the *maps* on sums and products are denoted $(+)$ and $(*)$.

$$
\begin{aligned}
&\mathit{map}\{\!|\mathsf{Char}|\!\} && = \mathit{id}\\
&\mathit{map}\{\!|\mathsf{Int}|\!\} && = \mathit{id}\\
&\mathit{map}\{\!|\mathsf{Unit}|\!\} && = \mathit{id}\\
&\mathit{map}\{\!|{:}{+}{:}|\!\}\ \mathit{mapa}\ \mathit{mapb} && = \mathit{mapa} + \mathit{mapb}\\
&\mathit{map}\{\!|{:}{*}{:}|\!\}\ \mathit{mapa}\ \mathit{mapb} && = \mathit{mapa} * \mathit{mapb}
\end{aligned}
$$

Even more succinctly, we have $\mathit{map}\{\!|{:}{+}{:}|\!\} = (+)$ and $\mathit{map}\{\!|{:}{*}{:}|\!\} = (*)$.

As usual, to apply a generic function we simply instantiate the type-index to a closed type.

$$
\begin{aligned}
&\mathit{map}\{\!|\mathsf{List\ Char}|\!\}\ \texttt{"hello world"}\\
&\Longrightarrow \texttt{"hello world"}\\
&\mathit{map}\{\!|\mathsf{List}|\!\}\ \mathit{toUpper}\ \texttt{"hello world"}\\
&\Longrightarrow \texttt{"HELLO WORLD"}
\end{aligned}
$$

We can also use *map* to define other generic functions.

$$
\begin{aligned}
&\mathit{distribute}\{\!|\mathsf{t} :: \star \to \star|\!\} && :: \forall \mathsf{a}\ \mathsf{b}\,.\,\mathsf{t}\ \mathsf{a} \to \mathsf{b} \to \mathsf{t}\ (\mathsf{a}, \mathsf{b})\\
&\mathit{distribute}\{\!|\mathsf{t}|\!\}\ x\ b && = \mathit{map}\{\!|\mathsf{t}|\!\}\ (\lambda a \to (a, b))\ x
\end{aligned}
$$

The call $\mathit{distribute}\{\!|\mathsf{t}|\!\}\ x\ b$ pairs the value b with every element contained in the structure x.

2.2 Kind-Indexed Types and Type-Indexed Values

In general, the definition of a type-indexed value consists of two parts: a type signature, which typically involves a kind-indexed type, and a set of equations, one for each type constant. A kind-indexed type is defined as follows:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Poly}\{\![\star]\!\}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \ldots\\
&\textbf{type}\ \mathsf{Poly}\{\![\kappa \to \nu]\!\}\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \forall \mathsf{a}_1\ \ldots\ \mathsf{a}_n\,.\,\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n\\
&\qquad\qquad\qquad\qquad\to \mathsf{Poly}\{\![\nu]\!\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ \ldots\ (\mathsf{t}_n\ \mathsf{a}_n).
\end{aligned}$$

The second clause is the same for all kind-indexed types so that the generic programmer merely has to fill out the right-hand side of the first equation. Actually, Generic Haskell offers a slightly more general form (see Section 2.4), which is the reason why we do not leave out the second clause.

Given a kind-indexed type, the definition of a type-indexed value takes on the following schematic form.

$$\begin{aligned}
&\mathit{poly}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}\\
&\mathit{poly}\{\!|\mathsf{Char}|\!\} &&= \ldots\\
&\mathit{poly}\{\!|\mathsf{Int}|\!\} &&= \ldots\\
&\mathit{poly}\{\!|\mathsf{Unit}|\!\} &&= \ldots\\
&\mathit{poly}\{\!|{:}{+}{:}|\!\}\ \mathit{polya}\ \mathit{polyb} &&= \ldots\\
&\mathit{poly}\{\!|{:}{*}{:}|\!\}\ \mathit{polya}\ \mathit{polyb} &&= \ldots
\end{aligned}$$

We have one clause for each primitive type (Int, Char etc) and one clause for each of the three type constructors Unit, ':*:', and ':+:'. Again, the generic programmer has to fill out the right-hand sides. To be well-typed, the $\mathit{poly}\{\!|\mathsf{t}{::}\kappa|\!\}$ instance must have type $\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$ as stated in the type signature of *poly*. Actually, the type signature can be more elaborate (we will see examples of this in Section 2.4).

The major insight of the mapping example is that a kind-indexed type can have several type arguments. Recall in this respect the type of the generic equality function:

$$\begin{aligned}
&\textbf{type}\ \mathsf{Eq}\{\![\star]\!\}\ \mathsf{t} &&= \mathsf{t} \to \mathsf{t} \to \mathsf{Bool}\\
&\textbf{type}\ \mathsf{Eq}\{\![\kappa \to \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a}\,.\,\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a} \to \mathsf{Eq}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a}).
\end{aligned}$$

Interestingly, we can generalize the type since the two arguments of equality need not be of the same type.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Eq}\{\![\star]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \mathsf{t}_1 \to \mathsf{t}_2 \to \mathsf{Bool}\\
&\textbf{type}\ \mathsf{Eq}\{\![\kappa \to \nu]\!\}\ \mathsf{t}_1\ \mathsf{t}_2 &&= \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Eq}\{\![\nu]\!\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)
\end{aligned}$$

We can assign $\mathit{eq}\{\!|\mathsf{t} :: \kappa|\!\}$ the more general type $\mathsf{Eq}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{t}$. Though this gives us a greater degree of flexibility, the definition of *eq* itself is not affected by this change! As an example, we could pass $\mathit{eq}\{\!|\mathsf{List}|\!\}$ the 'equality test' $\mathit{match} :: \mathsf{Female} \to \mathsf{Male} \to \mathsf{Bool}$ in order to check whether corresponding list entries match.

2.3 Embedding-Projection Maps

Most of the generic functions cannot sensibly be defined for the function space. For instance, equality of functions is not decidable. The mapping function *map* cannot be defined for function types since $(\rightarrow)$ is *contravariant* in its first argument:

$$\begin{aligned}
&(\rightarrow) && :: \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,(\mathsf{a}_2 \rightarrow \mathsf{a}_1) \rightarrow \\
& && \qquad\qquad \forall \mathsf{b}_1\ \mathsf{b}_2\,.\,(\mathsf{b}_1 \rightarrow \mathsf{b}_2) \rightarrow ((\mathsf{a}_1 \rightarrow \mathsf{b}_1) \rightarrow (\mathsf{a}_2 \rightarrow \mathsf{b}_2)) \\
&(f \rightarrow g)\ h = g \cdot h \cdot f.
\end{aligned}$$

In the case of mapping functions we can remedy the situation by drawing from the theory of embeddings and projections [13]. 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 (the code below make use of Haskell's record syntax).

$$\begin{aligned}
&\textbf{data}\ \mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 = EP\{from :: \mathsf{a}_1 \rightarrow \mathsf{a}_2, to :: \mathsf{a}_2 \rightarrow \mathsf{a}_1\} \\
&id_E && :: \forall \mathsf{a}\,.\,\mathsf{EP}\ \mathsf{a}\ \mathsf{a} \\
&id_E && = EP\{from = id, to = id\} \\
&(\circ) && :: \forall \mathsf{a}\ \mathsf{b}\ \mathsf{c}\,.\,\mathsf{EP}\ \mathsf{b}\ \mathsf{c} \rightarrow \mathsf{EP}\ \mathsf{a}\ \mathsf{b} \rightarrow \mathsf{EP}\ \mathsf{a}\ \mathsf{c} \\
&f \circ g && = EP\{from = from\ f \cdot from\ g, to = to\ g \cdot to\ f\}
\end{aligned}$$

Here, id_E 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, id_E and '$\circ$' give rise to the category $\mathcal{C}po^e$, the category of complete partial orders and embedding-projection pairs. This category has the interesting property that the function space can be turned into a *covariant* functor.

$$\begin{aligned}
&(+_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2\,.\,\mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 :\!+\!: \mathsf{b}_1)\ (\mathsf{a}_2 :\!+\!: \mathsf{b}_2) \\
&f +_E g && = EP\{from = from\ f + from\ g, to = to\ f + to\ g\} \\
&(*_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2\,.\,\mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 :\!*\!: \mathsf{b}_1)\ (\mathsf{a}_2 :\!*\!: \mathsf{b}_2) \\
&f *_E g && = EP\{from = from\ f * from\ g, to = to\ f * to\ g\} \\
&(\rightarrow_E) && :: \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{EP}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \forall \mathsf{b}_1\ \mathsf{b}_2\,.\,\mathsf{EP}\ \mathsf{b}_1\ \mathsf{b}_2 \rightarrow \mathsf{EP}\ (\mathsf{a}_1 \rightarrow \mathsf{b}_1)\ (\mathsf{a}_2 \rightarrow \mathsf{b}_2) \\
&f \rightarrow_E g && = EP\{from = to\ f \rightarrow from\ g, to = from\ f \rightarrow to\ g\}
\end{aligned}$$

Given these helper functions the generic embedding-projection map can be defined as follows.

$$\begin{aligned}
&\textbf{type}\ \mathsf{MapE}\{[\star]\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \mathsf{EP}\ \mathsf{t}_1\ \mathsf{t}_2 \\
&\textbf{type}\ \mathsf{MapE}\{[\kappa \rightarrow \nu]\}\ \mathsf{t}_1\ \mathsf{t}_2 && = \forall \mathsf{a}_1\ \mathsf{a}_2\,.\,\mathsf{MapE}\{[\kappa]\}\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow \mathsf{MapE}\{[\nu]\}\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)
\end{aligned}$$

$$\begin{aligned}
&mapE\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{MapE}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{t}\\
&mapE\{\!|\mathsf{Char}|\!\} &&= id_E\\
&mapE\{\!|\mathsf{Int}|\!\} &&= id_E\\
&mapE\{\!|\mathsf{Unit}|\!\} &&= id_E\\
&mapE\{\!|{:}{+}{:}|\!\} &&= (+_E)\\
&mapE\{\!|{:}{*}{:}|\!\} &&= (*_E)\\
&mapE\{\!|{\rightarrow}|\!\} &&= (\rightarrow_E)
\end{aligned}$$

We will see in Section 3.4 that embedding-projection maps are useful for changing the representation of data.

2.4 Reductions

The Haskell standard library defines a vast number of list processing functions. We have among others:

$$\begin{aligned}
&sum, product &&:: \forall \mathsf{a}\,.\,(Num\ \mathsf{a}) \Rightarrow [\mathsf{a}] \rightarrow \mathsf{a}\\
&and, or &&:: [\mathsf{Bool}] \rightarrow \mathsf{Bool}\\
&all, any &&:: \forall \mathsf{a}\,.\,(\mathsf{a} \rightarrow \mathsf{Bool}) \rightarrow ([\mathsf{a}] \rightarrow \mathsf{Bool})\\
&length &&:: \forall \mathsf{a}\,.\,[\mathsf{a}] \rightarrow \mathsf{Int}\\
&minimum, maximum &&:: \forall \mathsf{a}\,.\,(Ord\ \mathsf{a}) \Rightarrow [\mathsf{a}] \rightarrow \mathsf{a}\\
&concat &&:: \forall \mathsf{a}\,.\,[[\mathsf{a}]] \rightarrow [\mathsf{a}].
\end{aligned}$$

These are examples of so-called *reductions*. A reduction or a crush [31] is a function that collapses a structure of values of type t (such a structure is also known as a container) into a single value of type t. This section explains how to define reductions that work for all types of all kinds. To illustrate the main idea we first discuss three motivating examples. Let us start with a generic function that counts the number of values of type Int within a given structure of some type.

Here is the type of the generic counter

$$\begin{aligned}
&\textbf{type}\ \mathsf{Count}\{\![\star]\!\}\ \mathsf{t} &&= \mathsf{t} \rightarrow \mathsf{Int}\\
&\textbf{type}\ \mathsf{Count}\{\![\kappa \rightarrow \nu]\!\}\ \mathsf{t} &&= \forall \mathsf{a}\,.\,\mathsf{Count}\{\![\kappa]\!\}\ \mathsf{a} \rightarrow \mathsf{Count}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

and here is its definition.

$$\begin{aligned}
&count\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Count}\{\![\kappa]\!\}\ \mathsf{t}\\
&count\{\!|\mathsf{Char}|\!\}\ c &&= 0\\
&count\{\!|\mathsf{Int}|\!\}\ i &&= 1\\
&count\{\!|\mathsf{Unit}|\!\}\ Unit &&= 0\\
&count\{\!|{:}{+}{:}|\!\}\ counta\ countb\ (Inl\ a) &&= counta\ a\\
&count\{\!|{:}{+}{:}|\!\}\ counta\ countb\ (Inr\ b) &&= countb\ b\\
&count\{\!|{:}{*}{:}|\!\}\ counta\ countb\ (a\ {:}{*}{:}\ b) &&= counta\ a + countb\ b
\end{aligned}$$

Next, let us consider a slight variation: the function $sum\{\!|\mathsf{t}|\!\}$ defined below is identical to $count\{\!|\mathsf{t}|\!\}$ except for $\mathsf{t} = \mathsf{Int}$, in which case *sum* also returns 0.

$$
\begin{array}{ll}
sum\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Count}\{\![\kappa]\!\}\ \mathsf{t} \\
sum\{\!|\mathsf{Char}|\!\}\ c & = 0 \\
sum\{\!|\mathsf{Int}|\!\}\ i & = 0 \\
sum\{\!|\mathsf{Unit}|\!\}\ Unit & = 0 \\
sum\{\!|\mathsf{:+:}|\!\}\ suma\ sumb\ (Inl\ a) & = suma\ a \\
sum\{\!|\mathsf{:+:}|\!\}\ suma\ sumb\ (Inr\ b) & = sumb\ b \\
sum\{\!|\mathsf{:*:}|\!\}\ suma\ sumb\ (a :\!*\!: b) & = suma\ a + sumb\ b
\end{array}
$$

It is not hard to see that $sum\{\!|\mathsf{t}|\!\}\ x$ returns 0 for all types t of kind $\star$ (well, provided x is finite and fully defined). So one might be led to conclude that *sum* is not a very useful function. This conclusion is, however, too rash since *sum* can also be parameterized by type constructors. For instance, for unary type constructors *sum* has type

$$sum\{\!|\mathsf{t} :: \star \to \star|\!\} :: \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{Int}) \to (\mathsf{t\ a} \to \mathsf{Int})$$

If we pass the identity function to *sum*, we obtain a function that sums up a structure of integers. Another viable choice is *const* 1; this yields a function of type $\forall \mathsf{a} \,.\, \mathsf{t\ a} \to \mathsf{Int}$ that counts the number of values of type a in a given structure of type $\mathsf{t\ a}$.

$$
\begin{array}{l}
sum\{\!|\mathsf{List\ Int}|\!\}\ [2, 7, 1965] \\
\Longrightarrow 0 \\
sum\{\!|\mathsf{List}|\!\}\ id\ [2, 7, 1965] \\
\Longrightarrow 1974 \\
sum\{\!|\mathsf{List}|\!\}\ (const\ 1)\ [2, 7, 1965] \\
\Longrightarrow 3
\end{array}
$$

As usual, we can use generic abstractions to capture these idioms.

$$
\begin{array}{ll}
fsum\{\!|\mathsf{t} :: \star \to \star|\!\} & :: \mathsf{t\ Int} \to \mathsf{Int} \\
fsum\{\!|\mathsf{t}|\!\} & = sum\{\!|\mathsf{t}|\!\}\ id \\
fsize\{\!|\mathsf{t} :: \star \to \star|\!\} & :: \forall \mathsf{a} \,.\, \mathsf{t\ a} \to \mathsf{Int} \\
fsize\{\!|\mathsf{t}|\!\} & = sum\{\!|\mathsf{t}|\!\}\ (const\ 1)
\end{array}
$$

Using a similar approach we can flatten a structure into a list of elements. The type of the generic flattening function

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Flatten}\{\![\star]\!\}\ \mathsf{t\ x} & = \mathsf{t} \to [\mathsf{x}] \\
\textbf{type}\ \mathsf{Flatten}\{\![\kappa \to \nu]\!\}\ \mathsf{t\ x} & = \forall \mathsf{a} \,.\, \mathsf{Flatten}\{\![\kappa]\!\}\ \mathsf{a\ x} \to \mathsf{Flatten}\{\![\nu]\!\}\ (\mathsf{t\ a})\ \mathsf{x}
\end{array}
$$

makes use of a simple extension: $\mathsf{Flatten}\{\![\kappa]\!\}\ \mathsf{t\ x}$ takes an additional type parameter, x, that is passed unchanged to the base case. One can safely think of x as a type parameter that is global to the definition. The code for *flatten* is similar to the code for *sum*.

$$\begin{array}{ll}
\mathit{flatten}\{\!|\mathsf{t} :: \kappa|\!\} & :: \forall \mathsf{x}\,.\,\mathsf{Flatten}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{x} \\
\mathit{flatten}\{\!|\mathsf{Char}|\!\}\ c & = [\,] \\
\mathit{flatten}\{\!|\mathsf{Int}|\!\}\ i & = [\,] \\
\mathit{flatten}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} & = [\,] \\
\mathit{flatten}\{\!|{:}{+}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (\mathit{Inl}\ a) & = \mathit{fla}\ a \\
\mathit{flatten}\{\!|{:}{+}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (\mathit{Inr}\ b) & = \mathit{flb}\ b \\
\mathit{flatten}\{\!|{:}{*}{:}|\!\}\ \mathit{fla}\ \mathit{flb}\ (a :\!*\!: b) & = \mathit{fla}\ a +\!\!+\ \mathit{flb}\ b
\end{array}$$

The type signature of *flatten* makes precise that its instances are parametric in the type of list elements. We have, for instance,

$$\begin{array}{l}
\mathit{flatten}\{\!|\mathsf{Char}|\!\} :: \forall \mathsf{x}\,.\,\mathsf{Char} \to [\mathsf{x}] \\
\mathit{flatten}\{\!|\mathsf{Rose}|\!\} :: \forall \mathsf{x}\,.\,\forall \mathsf{a}\,.\,(\mathsf{a} \to [\mathsf{x}]) \to (\mathsf{Rose}\ \mathsf{a} \to [\mathsf{x}]).
\end{array}$$

Interestingly, the type dictates that $\mathit{flatten}\{\!|\mathsf{Char}|\!\} = \mathit{const}\ [\,]$. Like *sum*, the *flatten* function is pointless for types but useful for type constructors.

$$\begin{array}{ll}
\mathit{fflatten}\{\!|\mathsf{t} :: \star \to \star|\!\} & :: \forall \mathsf{a}\,.\,\mathsf{t}\ \mathsf{a} \to [\mathsf{a}] \\
\mathit{fflatten}\{\!|\mathsf{t}|\!\} & = \mathit{flatten}\{\!|\mathsf{t}|\!\}\ \mathit{wrap}\ \mathbf{where}\ \mathit{wrap}\ a = [a]
\end{array}$$

The definitions of *sum* 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. Its kind-indexed type is given by

$$\begin{array}{ll}
\mathbf{type}\ \mathsf{Reduce}\{\![\star]\!\}\ \mathsf{t}\ \mathsf{x} & = \mathsf{x} \to (\mathsf{x} \to \mathsf{x} \to \mathsf{x}) \to \mathsf{t} \to \mathsf{x} \\
\mathbf{type}\ \mathsf{Reduce}\{\![\kappa \to \nu]\!\}\ \mathsf{t}\ \mathsf{x} & = \forall \mathsf{a}\,.\,\mathsf{Reduce}\{\![\kappa]\!\}\ \mathsf{a}\ \mathsf{x} \to \mathsf{Reduce}\{\![\nu]\!\}\ (\mathsf{t}\ \mathsf{a})\ \mathsf{x}.
\end{array}$$

Note that the type argument x is passed unchanged to the recursive calls.

$$\begin{array}{ll}
\mathit{reduce}\{\!|\mathsf{t} :: \kappa|\!\} & :: \forall \mathsf{x}\,.\,\mathsf{Reduce}\{\![\kappa]\!\}\ \mathsf{t}\ \mathsf{x} \\
\mathit{reduce}\{\!|\mathsf{Char}|\!\}\ e\ \mathit{op}\ c & = e \\
\mathit{reduce}\{\!|\mathsf{Int}|\!\}\ e\ \mathit{op}\ i & = e \\
\mathit{reduce}\{\!|\mathsf{Unit}|\!\}\ e\ \mathit{op}\ \mathit{Unit} & = e \\
\mathit{reduce}\{\!|{:}{+}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (\mathit{Inl}\ a) & = \mathit{reda}\ e\ \mathit{op}\ a \\
\mathit{reduce}\{\!|{:}{+}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (\mathit{Inr}\ b) & = \mathit{redb}\ e\ \mathit{op}\ b \\
\mathit{reduce}\{\!|{:}{*}{:}|\!\}\ \mathit{reda}\ \mathit{redb}\ e\ \mathit{op}\ (a :\!*\!: b) & = \mathit{reda}\ e\ \mathit{op}\ a\ \text{`}\mathit{op}\text{`}\ \mathit{redb}\ e\ \mathit{op}\ b
\end{array}$$

Using *reduce* we can finally give generic versions of Haskell's list processing functions listed in the beginning of this section.

$$\begin{array}{ll}
\mathit{freduce}\{\!|\mathsf{t} :: \star \to \star|\!\} & :: \forall \mathsf{x}\,.\,\mathsf{x} \to (\mathsf{x} \to \mathsf{x} \to \mathsf{x}) \to \mathsf{t}\ \mathsf{x} \to \mathsf{x} \\
\mathit{freduce}\{\!|\mathsf{t}|\!\} & = \mathit{reduce}\{\!|\mathsf{t}|\!\}\ (\lambda e\ \mathit{op}\ a \to a)
\end{array}$$

$$
\begin{aligned}
&\mathit{fsum}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ 0\ (+)\\
&\mathit{fproduct}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ 1\ (*)\\
&\mathit{fand}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ \mathit{True}\ (\wedge)\\
&\mathit{for}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ \mathit{False}\ (\vee)\\
&\mathit{fall}\{\!|\mathsf{t}|\!\}\ f &&= \mathit{fand}\{\!|\mathsf{t}|\!\} \cdot \mathit{map}\{\!|\mathsf{t}|\!\}\ f\\
&\mathit{fany}\{\!|\mathsf{t}|\!\}\ f &&= \mathit{for}\{\!|\mathsf{t}|\!\} \cdot \mathit{map}\{\!|\mathsf{t}|\!\}\ f\\
&\mathit{fminimum}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ \mathit{maxBound}\ \mathit{min}\\
&\mathit{fmaximum}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ \mathit{minBound}\ \mathit{max}\\
&\mathit{fflatten}\{\!|\mathsf{t}|\!\} &&= \mathit{freduce}\{\!|\mathsf{t}|\!\}\ [\,]\ (+\!\!+)
\end{aligned}
$$

Typically, the two arguments of *freduce* form a *monoid*: the second argument is associative and has the first as its neutral element.

As an aside, note that the definition of *fflatten* has a quadratic running time. Exercise 8 seeks to remedy this defect.

2.5 Pretty Printing

Generic functions are defined by induction on the *structure* of types. Annoyingly, this is not quite enough. Consider, for example, the method *showsPrec* of the Haskell class *Show*. To be able to give a generic definition for *showsPrec*, the *names* of the constructors, and their fixities, must be made available.

To this end we provide one additional type case.

$$\mathit{poly}\{\!|\mathsf{Con}\ c|\!\}\ \mathit{polya} = \ldots$$

Roughly speaking, this case is invoked whenever we pass by a constructor. Quite unusual, the variable c that appears in the type index is bound to a *value* of type ConDescr and provides the required information about the name of a constructor, its arity etc.

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{ConDescr} &&= \mathit{ConDescr}\{\, \mathit{conName} :: \mathsf{String},\\
& && \qquad\qquad\quad \mathit{conType} :: \mathsf{String},\\
& && \qquad\qquad\quad \mathit{conArity} :: \mathsf{Int},\\
& && \qquad\qquad\quad \mathit{conLabels} :: \mathsf{Bool},\\
& && \qquad\qquad\quad \mathit{conFixity} :: \mathsf{Fixity}\}\\
&\mathbf{data}\ \mathsf{Fixity} &&= \mathit{Nonfix}\\
& &&\mid \mathit{Infix}\{\mathit{prec} :: \mathsf{Int}\}\\
& &&\mid \mathit{Infixl}\{\mathit{prec} :: \mathsf{Int}\}\\
& &&\mid \mathit{Infixr}\{\mathit{prec} :: \mathsf{Int}\}
\end{aligned}
$$

The *Con* data type itself is a simple wrapper type.

$$\mathbf{data}\ \mathsf{Con}\ \mathsf{a} = \mathit{Con}\ \mathsf{a}$$

Using *conName* and *conArity* we can implement a simple variant of Haskell's *showsPrec* function (ShowS, *shows*, *showChar*, and *showString* are predefined in Haskell).

```
type Shows{[⋆]} t              = t → ShowS
type Shows{[κ → ν]} t          = ∀a . Shows{[κ]} a → Shows{[ν]} (t a)
gshows{|t :: κ|}               :: Shows{[κ]} t
gshows{|:+:|} sa sb (Inl a)    = sa a
gshows{|:+:|} sa sb (Inr b)    = sb b
gshows{|Con c|} sa (Con a)
    | conArity c == 0          = showString (conName c)
    | otherwise                = showChar '(' · showString (conName c)
                                 · showChar ' ' · sa a · showChar ')'
gshows{|:*:|} sa sb (a :*: b)  = sa a · showChar ' ' · sb b
gshows{|Unit|} Unit            = showString ""
gshows{|Char|}                 = shows
gshows{|Int|}                  = shows
```

The first and the second equation discard the constructors *Inl* and *Inr*. They are not required since the constructor names can be accessed via the type pattern Con *c*. 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. The fourth equation applies if a constructor has more than one component. In this case the components are separated by a space.

In a nutshell, via ':+:' we get to the constructors, Con signals that we hit a constructor, and via ':*:' we get to the arguments of a constructor. Or, to put it differently, the generic programmer views, for instance, the list data type

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

as if it were given by the following type definition.

```
type List a = (Con Unit) :+: (Con (a :*: List a))
```

As a simple example, the list *Cons* 1 *Nil* is represented by *Inr* (*Con* (1 :*: *Inl* (*Con Unit*))).

It should be noted that descriptors of type *ConDescr* appear only in the type index; 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 Con *c*, then we tacitly assume that *poly*{|Con *c*|} *polya* = *polya*. (Actually, the default definition is slightly more involved since a and Con a are different types: the data constructor *Con* must be wrapped and unwrapped at the appropriate places. Section 3.4 explains how to accomplish this representation change in a systematic manner.) Now, why does the type Con *c* incorporate information about the constructor? 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 a generic version of *read*, which converts a string to a value. Consider the 'Con' case:

```
greads{|Con c|} ra s = [(x, s3) | (s1, s2) ← lex s,
                                  s1 == conName c,
                                  (x, s3) ← ra s2].
```

The important point is that *greads produces* (not consumes) the value, and yet it requires access to the constructor name.

The *gshows* function generates one long string, which does not look pretty at all when printed out. We can do better using Wadler's pretty printing combinators [48].

```
data Doc

empty  :: Doc
(◇)    :: Doc → Doc → Doc
string :: String → Doc
nl     :: Doc
nest   :: Int → Doc → Doc
group  :: Doc → Doc
render :: Int → Doc → String
```

The value *empty* represents the empty document, the operator '◇' catenates two documents, and *string* converts a string to an atomic document. The document *nl* denotes a potential line break. The function *nest* increases the indentation for all line breaks within its document argument. The function *group* marks its argument as a unit: it is printed out on a single line by converting all its potential line breaks into single spaces if this is possible without exceeding a given line-width limit. Finally, *render w d* converts a document to a string respecting the line width *w*.

The generic function *ppPrec*{|t|} *d x* takes a precedence level *d* (a value from 0 to 10), a value *x* of type *t* and returns a document of type Doc. The function essentially follows the structure of *gshows* except that it replaces the ShowS functions by pretty printing combinators.

```
type Pretty{[⋆]} t                   = Int → t → Doc
type Pretty{[κ → ν]} t               = ∀a . Pretty{[κ]} a → Pretty{[ν]} (t a)
ppPrec{|t :: κ|}                     :: Pretty{[κ]} t
ppPrec{|:+:|} ppa ppb d (Inl a)      = ppa d a
ppPrec{|:+:|} ppa ppb d (Inr b)      = ppb d b
ppPrec{|Con c|} ppa d (Con a)
    | conArity c == 0                = string (conName c)
    | otherwise                      = group (nest 2 (ppParen (d > 9) doc))
  where doc                          = string (conName c) ◇ nl ◇ ppa 10 a
ppPrec{|:*:|} ppa ppb d (a :*: b)    = ppa d a ◇ nl ◇ ppb d b
ppPrec{|Unit|} d Unit                = empty
```

```
ppPrec{|Int|} d i    = string (show i)
ppPrec{|Char|} d c   = string (show c)
ppParen              :: Bool → Doc → Doc
ppParen False d      = d
ppParen True d       = string "(" ◇ d ◇ string ")"
```

The helper function *ppParen* encloses its second argument in parenthesis if its first evaluates to *True*.

2.6 Running Generic Haskell

This section explains how to use the Generic Haskell compiler called `gh`. It has two basic modes of operation. If `gh` is called without arguments, the user is prompted for a file name (relative to the working directory). The compiler then processes the file and generates an output file with the same basename as the input file, but the extension '`.hs`'. Generic Haskell source files typically have the extension '`.ghs`'. Alternatively, input files can be specified on the command line. A typical invocation is:

```
path_to_gh/bin/gh -L path_to_gh/lib/ your_file.ghs
```

A number of command line options are available:

```
 Usage: gh [options...] files...
   -v      --verbose      (number of v's controls the verbosity)
   -m      --make         follow dependencies
   -V      --version      show version info
   -h, -?  --help         show help
           --cut=N        cut computations after N iterations
   -C      --continue     continue after errors
   -L DIR  --library=DIR  add DIR to search path
```

The first level of verbosity (no `-v` flag) produces only error messages. The second level (`-v`) additionally provides diagnostic information and warnings. The third level (`-vv`) produces debugging information.

The `-m` (or `--make`) option instructs the compiler to chase module dependencies and to automatically process those modules which require compilation.

The option `--cut=N` stops the compilation after `N` iterations of the specialization mechanism (the default is to stop after 50 iterations). The first level of verbosity (`-v`) can be used to report the number of required iterations.

The `-C` (or `--continue`) option forces the compiler to continue compilation even when an error is encountered. This can be used to generate further error messages or to inspect the generated code.

The Generic Haskell compiler compiles '`.ghs`' files and produces '`.hs`' files which can subsequently be compiled using a Haskell compiler. In addition, the compiler produces '`.ghi`' interface files, which will be used in subsequent compilations to avoid unnecessary recompilation. As an example, Figure 1 displays the source code for the generic version of Haskell's *shows* function. Note that *type* arguments are enclosed in `{|·|}` brackets, while `{[·]}` embraces *kind* arguments.

The Generic Haskell compiler generates ordinary Haskell code (Haskell 98 augmented with rank-2 types), which can be run or compiled using the Glasgow Haskell Compiler, Hugs, or any other Haskell compiler. You only have to ensure

```
type Shows {[ * ]} t    =  t -> ShowS
type Shows {[ k -> l ]} t
       =  forall a . Shows {[ k ]} a -> Shows {[ l ]} (t a)

gshows {| t :: k |}     :: Shows {[ k ]} t
gshows {| :+: |} sa sb (Inl a)  =  sa a
gshows {| :+: |} sa sb (Inr b)  =  sb b
gshows {| Con c|} sa (Con a)
  | conArity c == 0     =  showString (conName c)
  | otherwise           =  showChar '(' . showString (conName c)
                           . showChar ' ' . sa a . showChar ')'
gshows {| :*: |} sa sb (a :*: b)
                        =  sa a . showChar ' ' . sb b
gshows {| Unit |} Unit  =  showString ""
gshows {| Char |}       =  shows
gshows {| Int |}        =  shows

data Tree a b           =  Tip a | Node (Tree a b) b (Tree a b)

main                    =  putStrLn (gshows {| Tree Int String |} ex "")

ex                      :: Tree Int String
ex                      =  Node (Tip 1) "hello"
                                (Node (Tip 2) "world" (Tip 3))
```

Fig. 1. A generic implementation of the *shows* function.

that the path to GHPrelude.hs (and to other Generic Haskell libraries), which can be found in the lib subdirectory, is included in your compiler's search path.

The Generic Haskell compiler is shipped with an extensive library, which provides further examples of generic functions.

Exercise 6. Let M be a monad. A *monadic mapping function* for a type constructor F of kind $\star \to \star$ lifts a given function of type $\mathsf{a} \to \mathsf{M}\ \mathsf{b}$ to a function of type $\mathsf{F}\ \mathsf{a} \to \mathsf{M}\ (\mathsf{F}\ \mathsf{b})$. Define a generic version of the monadic map. First define a kind-indexed type and then give equations for each of the type constants.

The standard mapping function is essentially determined by its type (each equation is more or less inevitable). Does this also hold for monadic maps?

Exercise 7. 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 $\mathsf{List}\ \mathsf{a}_1$ and $\mathsf{List}\ \mathsf{a}_2$ and pairs corresponding elements producing a list of type $\mathsf{List}\ (\mathsf{a}_1, \mathsf{a}_2)$. Define a generic version of *zip* that works for all types of all kinds. *Hint:* the kind-indexed type of *zip* is essentially a three parameter variant of Map.

Exercise 8. The implementation of *fflatten* given in Section 2.4 has 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* [6] one can improve the running time to $O(n)$. This technique can be captured using a generic function, called a *right reduction*. Its kind-indexed type is given by

$$\textbf{type}\ \mathsf{Reducer}\{[\star]\}\ \mathsf{t}\ \mathsf{x} = \mathsf{t} \to \mathsf{x} \to \mathsf{x}$$
$$\textbf{type}\ \mathsf{Reducer}\{[\kappa \to \nu]\}\ \mathsf{t}\ \mathsf{x} = \forall \mathsf{a}\,.\,\mathsf{Reducer}\{[\kappa]\}\ \mathsf{a}\ \mathsf{x} \to \mathsf{Reducer}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})\ \mathsf{x}$$

The second argument of type x is the accumulator. Fill in the definition of *reducer* and define *fflatten* in terms of *reducer*. Why is *reducer* called a right reduction? Also define its dual, a *left reduction*.

Exercise 9. The generic function *fsize* computes the size of a container type. Can you define a function that computes its height? *Hint:* you have to make use of the constructor case Con.

Exercise 10 (this may take some time). In Section 1.3 we have implemented a simple form of data compression. A more sophisticated scheme could use Huffman coding, emitting variable-length codes for the constructors. Implement a function that given a sample element of a data type counts the number of occurrences of each constructor. *Hint:* consider the type ConDescr. Which information is useful for your task?

3 Generic Haskell—Theory

This section highlights the theory underlying Generic Haskell (most of the material is taken from [22]).

We have already indicated that Generic Haskell takes a transformational approach to generic programming: a generic function is translated into a family of polymorphic functions. We will see in this section that this transformation can be phrased as an interpretation of the simply typed lambda calculus (types are simply typed lambda terms with kinds playing the rôle of types). To make this idea precise we switch from Haskell to the polymorphic lambda calculus. The simply typed lambda calculus and the polymorphic lambda calculus are introduced in Sections 3.1 and 3.2, respectively. Section 3.3 then shows how to specialize generic functions to instances of data types.

The polymorphic lambda calculus is the language of choice for the theoretical treatment of generic definitions. Though Haskell comes quite close to this ideal language, there is one fundamental difference between Haskell and (our presentation) of the polymorphic lambda calculus. In Haskell, each data declaration introduces a *new* type; two types are equal iff they have the same name. By contrast, in the polymorphic lambda calculus two types are equal iff they have the same structure. Section 3.4 explains how to adapt the technique of Section 3.3 to a type system based on name equivalence.

3.1 The Simply Typed Lambda Calculus as a Type Language

This section introduces the language of kinds and types that we will use in the sequel. The type system is essentially that of Haskell smoothing away some of its irregularities. We have see in Section 1.2 that Haskell offers one basic construct for defining new types: data type declarations.

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}.$$

From a language design point of view the **data** construct is quite a beast as it combines no less than four different features: type abstraction, type recursion, n-ary sums, and n-ary products. The types on the right-hand side are built from type constants (that is, primitive type constructors), type variables, and type application. Thus, Haskell's type system essentially corresponds to the simply typed lambda calculus with kinds playing the rôle of types.

In the sequel we review the syntax and the semantics of the simply typed lambda calculus. A basic knowledge of this material will prove useful when we discuss the specialization of type-indexed values. Most of the definitions are taken from the excellent textbook by J. Mitchell [34].

Syntax. The simply typed lambda calculus has a two-level structure: kinds and types (since we will use the calculus to model Haskell's type system we continue to speak of kinds and types).

kind terms	$\kappa, \nu \in \mathsf{Kind}$
type constants	$\mathsf{C}, \mathsf{D} \in \mathsf{Const}$
type variables	$\mathsf{a}, \mathsf{b} \in \mathsf{Var}$
type terms	$\mathsf{t}, \mathsf{u} \in \mathsf{Type}$

Note that we use Greek letters for kinds and Sans Serif letters for types.

Kind terms are formed according to the following grammar.

$$\begin{array}{rll} \kappa, \nu \in \mathsf{Kind} ::= & \star & \text{kind of types} \\ \mid & (\kappa \to \nu) & \text{function kind} \end{array}$$

As usual, we assume that '$\to$' associates to the right.

Pseudo-type terms are built from type constants and type variables using type abstraction and type application.

$$\begin{array}{rll} \mathsf{t}, \mathsf{u} \in \mathsf{Type} ::= & \mathsf{C} & \text{type constant} \\ \mid & \mathsf{a} & \text{type variable} \\ \mid & (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) & \text{type abstraction} \\ \mid & (\mathsf{t}\ \mathsf{u}) & \text{type application} \end{array}$$

We assume that type abstraction extends as far to the right as possible and that type application associates to the left. For typographic simplicity, we will often omit the kind annotation in $\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}$ (especially if $\nu = \star$). Finally, we abbreviate nested abstractions $\Lambda \mathsf{a}_1 \,.\, \ldots \Lambda \mathsf{a}_m \,.\, \mathsf{t}$ by $\Lambda \mathsf{a}_1\ \ldots\ \mathsf{a}_m \,.\, \mathsf{t}$.

The choice of Const, the set of type constants, is more or less arbitrary. However, in order to model Haskell's data declarations we assume that Const comprises at least the constants Char, Int, Unit, ':+:', ':*:', and a family of fixed point operators:

$$\begin{aligned}\mathsf{Const} \supseteq\ & \{\mathsf{Char} :: \star, \mathsf{Int} :: \star, \mathsf{Unit} :: \star, (:\!*\!:) :: \star \to \star \to \star, (:\!+\!:) :: \star \to \star \to \star\} \\ & \cup \{\mathsf{Fix}_\kappa :: (\kappa \to \kappa) \to \kappa \mid \kappa \in \mathsf{Kind}\}.\end{aligned}$$

As usual, we write binary type constants infix. We assume that ':*:' and ':+:' associate to the right and that ':*:' binds more tightly than ':+:'. The set of type constants includes a family of fixed point operators indexed by kind. In the examples, we will often omit the kind annotation in Fix_κ.

In order to define 'well-kinded' type terms we need the notion of a context. A context is a finite set of kind assumptions of the form $\mathsf{a} :: \kappa$. It is convenient to view a context Γ as a finite map from type variables to kinds and write $dom(\Gamma)$ for its domain. Likewise, we view Const as a finite map from type constants to kinds. A pseudo-type term t is called a *type term* if there is a context Γ and a kind κ such that $\Gamma \vdash \mathsf{t} :: \kappa$ is derivable using the rules depicted in Figure 2.

$$\frac{}{\Gamma \vdash \mathsf{C} :: \mathsf{Const}(\mathsf{C})}\ (\text{T-CONST})$$

$$\frac{}{\Gamma \vdash \mathsf{a} :: \Gamma(\mathsf{a})}\ (\text{T-VAR})$$

$$\frac{\Gamma, \mathsf{a} :: \nu \vdash \mathsf{t} :: \kappa}{\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) :: (\nu \to \kappa)}\ (\text{T-}\!\to\!\text{-INTRO})$$

$$\frac{\Gamma \vdash \mathsf{t} :: (\nu \to \kappa) \qquad \Gamma \vdash \mathsf{u} :: \nu}{\Gamma \vdash (\mathsf{t}\ \mathsf{u}) :: \kappa}\ (\text{T-}\!\to\!\text{-ELIM})$$

Fig. 2. Kinding rules.

The equational proof system of the simply typed lambda calculus is given by the rules in Figure 3. Let $\mathcal{E}$ be a possibly empty set of equations between type terms. We write $\Gamma \vdash_{\mathcal{E}} \mathsf{t}_1 = \mathsf{t}_2 :: \kappa$ to mean that the type equation $\mathsf{t}_1 = \mathsf{t}_2$ is provable using the rules and the equations in $\mathcal{E}$.

$$\frac{}{\Gamma \vdash ((\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t})\ \mathsf{u} = \mathsf{t}[\mathsf{a} := \mathsf{u}]) :: \kappa}\ (\text{T-}\beta)$$

$$\frac{\mathsf{a}\ \text{not free in}\ \mathsf{t}}{\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}\ \mathsf{a} = \mathsf{t}) :: (\nu \to \kappa)}\ (\text{T-}\eta)$$

$$\frac{}{\Gamma \vdash (\mathsf{Fix}_\kappa\ \mathsf{t} = \mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})) :: \kappa}\ (\text{T-FIX})$$

Fig. 3. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

The equational proof rules identify a recursive type $\mathsf{Fix}_\kappa\ \mathsf{t}$ and its unfolding $\mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})$. In general, there are two flavours of recursive types: equi-recursive types and iso-recursive types, see [12]. In the latter system $\mathsf{Fix}_\kappa\ \mathsf{t}$ and $\mathsf{t}\ (\mathsf{Fix}_\kappa\ \mathsf{t})$ must only be isomorphic rather than equal. The subsequent development is largely independent of this design choice. We use equi-recursive types because they simplify the presentation somewhat.

Modelling Data Declarations. Using the simply typed lambda calculus as a type language we can easily translate data type declarations into type terms. For instance, the type B defined by the schematic data declaration in the beginning of this section is modelled by (we tacitly assume that the kinds of the type variables have been inferred)

$$\mathsf{Fix}\ (\Lambda \mathsf{B}\,.\,\Lambda \mathsf{a}_1\ \ldots\ \mathsf{a}_m\,.\,(\mathsf{t}_{11} :\!*\!: \cdots :\!*\!: \mathsf{t}_{1m_1}) :\!+\!: \cdots :\!+\!: (\mathsf{t}_{n1} :\!*\!: \cdots :\!*\!: \mathsf{t}_{nm_n})),$$

where $\mathsf{t}_1 :\!*\!: \cdots :\!*\!: \mathsf{t}_k = \mathsf{Unit}$ for $k = 0$. For simplicity, n-ary sums are reduced to binary sums and n-ary products to binary products. For instance, the data declaration

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

is translated to

$$\mathsf{Fix}\ (\Lambda \mathsf{List}\,.\,\Lambda \mathsf{a}\,.\,\mathsf{Unit} :\!+\!: \mathsf{a} :\!*\!: \mathsf{List}\ \mathsf{a}).$$

Interestingly, the representation of regular types such as List can be improved by applying a technique called *lambda-dropping* [11]: if $\mathsf{Fix}\ (\Lambda \mathsf{f}\,.\,\Lambda \mathsf{a}\,.\,\mathsf{t})$ is regular, then it is equivalent to $\Lambda \mathsf{a}\,.\,\mathsf{Fix}\ (\Lambda \mathsf{b}\,.\,\mathsf{t}[\mathsf{f}\ \mathsf{a} := \mathsf{b}])$ where $\mathsf{t}[\mathsf{t}_1 := \mathsf{t}_2]$ denotes the type term, in which all occurrences of t_1 are replaced by t_2. For instance, the λ-dropped version of $\mathsf{Fix}\ (\Lambda \mathsf{List}\,.\,\Lambda \mathsf{a}\,.\,\mathsf{Unit} :\!+\!: \mathsf{a} :\!*\!: \mathsf{List}\ \mathsf{a})$ is $\Lambda \mathsf{a}\,.\,\mathsf{Fix}\ (\Lambda \mathsf{b}\,.\,\mathsf{Unit} :\!+\!: \mathsf{a} :\!*\!: \mathsf{b})$. The λ-dropped version employs the fixed point operator at kind $\star$ whereas the original, the so-called λ-*lifted* version employs the fixed point operator at kind $\star \to \star$. Nested types such as Sequ 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, [8].

Environment Models. This section is concerned with the denotational semantics of the simply typed lambda 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.

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. An applicative structure is similar to an algebra, in that we need a set of 'values' for each kind and an interpretation for each type constant. Additionally, we have to give meaning to type abstraction and type application.

Definition 1. *An applicative structure $\mathcal{A}$ is a tuple* $(\mathbf{A}, \mathbf{app}, \mathbf{const})$ *such that*

- $\mathbf{A} = (\mathbf{A}^{\kappa} \mid \kappa \in \mathsf{Kind})$ *is a family of sets,*
- $\mathbf{app} = (\mathbf{app}_{\kappa,\nu} : \mathbf{A}^{\kappa\to\nu} \to (\mathbf{A}^{\kappa} \to \mathbf{A}^{\nu}) \mid \kappa, \nu \in \mathsf{Kind})$ *is a family of maps, and*
- $\mathbf{const} : \mathsf{Const} \to \mathbf{A}$ *is a mapping from type constants to values such that* $\mathbf{const}(\mathsf{C}) \in \mathbf{A}^{\mathsf{Const}(\mathsf{C})}$ *for every* $\mathsf{C} \in \mathit{dom}(\mathsf{Const})$.

The first condition on models requires that equality between elements of function kinds is standard equality on functions.

Definition 2. *An applicative structure* $\mathcal{A} = (\mathbf{A}, \mathbf{app}, \mathbf{const})$ *is extensional, if* $\forall \phi_1, \phi_2 \in \mathbf{A}^{\kappa\to\nu} \,.\, (\forall \alpha \in \mathbf{A}^{\kappa} \,.\, \mathbf{app}\ \phi_1\ \alpha = \mathbf{app}\ \phi_2\ \alpha) \supset \phi_1 = \phi_2$.

A simple example for an applicative structure is the set of type terms itself: Let $\mathcal{H}$ be an infinite context that provides infinitely many type variables of each kind. An extensional applicative structure $(\mathsf{Type}, \mathbf{app}, \mathbf{const})$ may then be defined by letting $\mathsf{Type}^{\kappa} = \{\mathsf{t} \mid \Gamma \vdash \mathsf{t} :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H}\}$, $\mathbf{app}\ \mathsf{t}\ \mathsf{u} = (\mathsf{t}\ \mathsf{u})$, and $\mathbf{const}(\mathsf{C}) = \mathsf{C}$.

The second condition on models ensures that the applicative structure has enough points so that every type term containing type 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 from type variables to values. If Γ is a context, then we say η satisfies Γ if $\eta(\mathsf{a}) \in \mathbf{A}^{\Gamma(\mathsf{a})}$ for every $\mathsf{a} \in \mathit{dom}(\Gamma)$. If η is an environment, then $\eta(\mathsf{a} := \alpha)$ is the environment mapping a to α and b to $\eta(\mathsf{b})$ for b different from a.

Definition 3. *An applicative structure* $\mathcal{A} = (\mathbf{A}, \mathbf{app}, \mathbf{const})$ *is an environment model if it is extensional and if the clauses below define a total meaning function on terms* $\Gamma \vdash \mathsf{t} :: \kappa$ *and environments such that* η *satisfies* Γ.

$$\begin{aligned}
&\mathcal{A}[\![\Gamma \vdash \mathsf{C} :: \mathsf{Const}(\mathsf{C})]\!]\eta = \mathbf{const}(\mathsf{C})\\
&\mathcal{A}[\![\Gamma \vdash \mathsf{a} :: \Gamma(\mathsf{a})]\!]\eta \qquad = \eta(\mathsf{a})\\
&\mathcal{A}[\![\Gamma \vdash (\Lambda \mathsf{a} :: \nu \,.\, \mathsf{t}) :: (\nu \to \kappa)]\!]\eta\\
&\qquad\qquad\qquad\qquad\quad = \textit{the unique } \phi \in \mathbf{A}^{\nu\to\kappa} \textit{ such that for all } \alpha \in \mathbf{A}^{\nu}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{app}_{\nu,\kappa}\ \phi\ \alpha = \mathcal{A}[\![\Gamma, \mathsf{a} :: \nu \vdash \mathsf{t} :: \kappa]\!]\eta(\mathsf{a} := \alpha)\\
&\mathcal{A}[\![\Gamma \vdash (\mathsf{t}\ \mathsf{u}) :: \kappa]\!]\eta \qquad = \mathbf{app}_{\nu,\kappa}\ (\mathcal{A}[\![\Gamma \vdash \mathsf{t} :: (\nu \to \kappa)]\!]\eta)\ (\mathcal{A}[\![\Gamma \vdash \mathsf{u} :: \nu]\!]\eta)
\end{aligned}$$

Note that extensionality guarantees the uniqueness of the element ϕ whose existence is postulated in the third clause.

The set of type terms can be turned into an environment model if we identify type terms that are provably equal. Let $\mathcal{E}$ be a possibly empty set of equations between type terms. Then we define the equivalence class of types $[\mathsf{t}] = \{\, T' \mid \Gamma \vdash_{\mathcal{E}} \mathsf{t} = T' :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H}\,\}$ and let $\mathsf{Type}^{\kappa}/\mathcal{E} = \{[\mathsf{t}] \mid \mathsf{t} \in \mathsf{Type}^{\kappa}\}$, $(\mathbf{app}/\mathcal{E})\ [\mathsf{t}]\ [\mathsf{u}] = [\mathsf{t}\ \mathsf{u}]$, and $(\mathbf{const}/\mathcal{E})\ (\mathsf{C}) = [\mathsf{C}]$. Then the applicative structure $(\mathsf{Type}/\mathcal{E}, \mathbf{app}/\mathcal{E}, \mathbf{const}/\mathcal{E})$ is an environment model.

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

$$\frac{\Gamma \vdash \mathsf{r} :: \star \qquad \Gamma \vdash \mathsf{s} :: \star}{\Gamma \vdash (\mathsf{r} \to \mathsf{s}) :: \star} \ (\text{T-FUN})$$

$$\frac{\Gamma, \mathsf{a} :: \nu \vdash \mathsf{s} :: \star}{\Gamma \vdash (\forall \mathsf{a} :: \nu \,.\, \mathsf{s}) :: \star} \ (\text{T-ALL})$$

Fig. 4. Additional kinding rules for type schemes.

Definition 4. *An applicative structure* $\mathcal{A} = (\mathbf{A}, \mathbf{app}, \mathbf{const})$ *satisfies the combinatory model condition if for all kinds* κ, ν *and* μ *there exist elements* $\mathbf{K}_{\kappa,\nu} \in \mathbf{A}^{\kappa\to\nu\to\kappa}$ *and* $\mathbf{S}_{\kappa,\nu,\mu} \in \mathbf{A}^{(\kappa\to\nu\to\mu)\to(\kappa\to\nu)\to\kappa\to\mu}$ *such that*

$$\begin{aligned}
&\mathbf{app}\ (\mathbf{app}\ \mathbf{K}\ \mathsf{a})\ \mathsf{b} && = \mathsf{a} \\
&\mathbf{app}\ (\mathbf{app}\ (\mathbf{app}\ \mathbf{S}\ \mathsf{a})\ \mathsf{b})\ \mathsf{c} && = \mathbf{app}\ (\mathbf{app}\ \mathsf{a}\ \mathsf{c})\ (\mathbf{app}\ \mathsf{b}\ \mathsf{c})
\end{aligned}$$

for all a, b *and* c *of the appropriate kinds.*

3.2 The Polymorphic Lambda Calculus

We have seen in Section 1.3 that instances of generic functions require first-class polymorphism. For instance, *eqGRose* takes a polymorphic argument to a polymorphic result. To make the use of polymorphism explicit we will use a variant of the polymorphic lambda calculus [14] (also known as $F\omega$) both for defining and for specializing type-indexed values. This section provides a brief introduction to the calculus. As an aside, note that a similar language is also used as the internal language of the Glasgow Haskell Compiler [39].

The polymorphic lambda calculus has a three-level structure (kinds, type schemes, and terms) incorporating the simply typed lambda calculus on the type level.

type schemes	$\mathsf{r}, \mathsf{s} \in \mathsf{Scheme}$
individual constants	$c, d \in const$
individual variables	$a, b \in var$
terms	$t, u \in term$

We use Roman letters for terms.

Pseudo-type schemes are formed according to the following grammar.

$$\begin{array}{lll}
\mathsf{r}, \mathsf{s} \in \mathsf{Scheme} ::= & \mathsf{t} & \text{type term} \\
| & (\mathsf{r} \to \mathsf{s}) & \text{function type} \\
| & (\forall \mathsf{a} :: \nu \,.\, \mathsf{s}) & \text{polymorphic type}
\end{array}$$

A pseudo-type scheme s is called a type scheme if there is a context Γ and a kind κ such that $\Gamma \vdash \mathsf{s} :: \kappa$ is derivable using the rules listed in Figures 2 and 4.

Pseudo-terms are given by the grammar

$$\frac{}{\Gamma \vdash c :: \mathit{const}(c)}\ (\textsc{var})$$

$$\frac{}{\Gamma \vdash a :: \Gamma(a)}\ (\textsc{const})$$

$$\frac{\Gamma, a :: \mathsf{s} \vdash t :: \mathsf{r}}{\Gamma \vdash (\lambda a :: \mathsf{s} \,.\, t) :: (\mathsf{s} \to \mathsf{r})}\ (\to\textsc{-intro})$$

$$\frac{\Gamma \vdash t :: (\mathsf{s} \to \mathsf{r}) \qquad \Gamma \vdash u :: \mathsf{s}}{\Gamma \vdash (t\ u) :: \mathsf{r}}\ (\to\textsc{-elim})$$

$$\frac{\Gamma, \mathsf{a} :: \nu \vdash t :: \mathsf{s}}{\Gamma \vdash (\lambda \mathsf{a} :: \nu \,.\, t) :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s})}\ (\forall\textsc{-intro})$$

$$\frac{\Gamma \vdash t :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s}) \qquad \Gamma \vdash \mathsf{r} :: \nu}{\Gamma \vdash (t\ \mathsf{r}) :: \mathsf{s}[\mathsf{a} := \mathsf{r}]}\ (\forall\textsc{-elim})$$

$$\frac{\Gamma \vdash t :: \mathsf{r} \qquad \Gamma \vdash (\mathsf{r} = \mathsf{s}) :: \star}{\Gamma \vdash t :: \mathsf{s}}\ (\textsc{conv})$$

Fig. 5. Typing rules.

$$\begin{array}{rcll}
t, u \in \mathit{term} & ::= & c & \text{constant} \\
& \mid & a & \text{variable} \\
& \mid & (\lambda a :: \mathsf{s} \,.\, t) & \text{abstraction} \\
& \mid & (t\ u) & \text{application} \\
& \mid & (\lambda \mathsf{a} :: \nu \,.\, t) & \text{universal abstraction} \\
& \mid & (t\ \mathsf{r}) & \text{universal application.}
\end{array}$$

Here, $\lambda \mathsf{a} :: \nu \,.\, t$ denotes universal abstraction (forming a polymorphic value) and $t\ \mathsf{r}$ denotes universal application (instantiating a polymorphic value). Note that we use the same syntax for value abstraction $\lambda a :: \mathsf{s} \,.\, t$ (here a is a value variable) and universal abstraction $\lambda \mathsf{a} :: \nu \,.\, t$ (here a is a type variable). We assume that the set *const* of value constants includes at least the polymorphic fixed point operator

$$\mathit{fix} :: \forall \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a}) \to \mathsf{a}$$

and suitable functions for each of the other type constants C in $\mathit{dom}(\mathsf{Const})$ (such as *Unit* for 'Unit', *Inl*, *Inr*, and **case** for ':+:', and *outl*, *outr*, and (_ :*: _) for ':*:'). To improve readability we will usually omit the type argument of *fix*.

To give the typing rules we have to extend the notion of context. A context is a finite set of kind assumptions $\mathsf{a} :: \kappa$ and type assumptions $a :: \mathsf{s}$. We say a context Γ is closed if Γ is either empty, or if $\Gamma = \Gamma_1, \mathsf{a} :: \kappa$ with Γ_1 closed, or if $\Gamma = \Gamma_1, a :: \mathsf{s}$ with Γ_1 closed and $\mathit{free}(\mathsf{s}) \subseteq \mathit{dom}(\Gamma_1)$. In the following we assume that contexts are closed. This restriction is necessary to prevent non-sensible terms such as $a :: \mathsf{a} \vdash \lambda \mathsf{a} :: \star \,.\, a$ where the value variable a carries the type variable a out of scope. A pseudo-term t is called a term if there is some context Γ and some type scheme s such that $\Gamma \vdash t :: \mathsf{s}$ is derivable using the typing rules depicted in Figure 5. Note that rule (CONV) allows us to interchange provably equal types.

$$\frac{}{\Gamma \vdash ((\lambda a :: \mathsf{s} \,.\, t)\ u = t[a := u]) :: \mathsf{r}}\ (\beta)$$

$$\frac{a \text{ not free in } t}{\Gamma \vdash (\lambda a :: \mathsf{s} \,.\, t\ a = t) :: (\mathsf{s} \to \mathsf{r})}\ (\eta)$$

$$\frac{}{\Gamma \vdash ((\lambda \mathsf{a} :: \nu \,.\, t)\ \mathsf{r} = t[\mathsf{a} := \mathsf{r}]) :: \mathsf{s}[\mathsf{a} := \mathsf{r}]}\ (\beta)_\forall$$

$$\frac{\mathsf{a} \text{ not free in } t}{\Gamma \vdash (\lambda \mathsf{a} :: \nu \,.\, t\ \mathsf{a} = t) :: (\forall \mathsf{a} :: \nu \,.\, \mathsf{s})}\ (\eta)_\forall$$

$$\frac{}{\Gamma \vdash (\mathit{fix}\ \mathsf{r}\ f = f\ (\mathit{fix}\ \mathsf{r}\ f)) :: \mathsf{r}}\ (\textsc{fix})$$

Fig. 6. Equational proof rules (the usual 'logical' rules for reflexivity, symmetry, transitivity, and congruence are omitted).

The equational proof system of the polymorphic lambda calculus is given by the rules in Figure 6. When we discuss the specialization of type-indexed values, we will consider type schemes and terms modulo provable equality. Let $\mathcal{H}$ be an infinite context that provides type variables of each kind and variables of each type scheme and let $\mathcal{E}$ be a set of equations between type schemes and/or between terms. Analogous to the entities Type^κ, $[\mathsf{t}]$ and $\mathsf{Type}^\kappa/\mathcal{E}$ we define $\mathsf{Scheme}^\kappa = \{\, \mathsf{s} \mid \Gamma \vdash \mathsf{s} :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H} \,\}$, the equivalence class $[\mathsf{s}] = \{\, \mathsf{s}' \mid \Gamma \vdash_{\mathcal{E}} \mathsf{s} = \mathsf{s}' :: \kappa$ for some finite $\Gamma \subseteq \mathcal{H} \,\}$, $\mathsf{Scheme}^\kappa/\mathcal{E} = \{\, [\mathsf{s}] \mid \mathsf{s} \in \mathsf{Scheme}^\kappa \,\}$, and $\mathit{Term}^{\mathsf{s}} = \{\, t \mid \Gamma \vdash t :: \mathsf{s}$ for some finite $\Gamma \subseteq \mathcal{H} \,\}$, $[t] = \{\, t' \mid \Gamma \vdash_{\mathcal{E}} t = t' :: \mathsf{s}$ for some finite $\Gamma \subseteq \mathcal{H} \,\}$, and $\mathit{Term}^{\mathsf{s}}/\mathcal{E} = \{\, [t] \mid t \in \mathit{Term}^{\mathsf{s}} \,\}$. Note that $[\mathsf{s}_1] = [\mathsf{s}_2]$ implies $\mathit{Term}^{\mathsf{s}_1} = \mathit{Term}^{\mathsf{s}_2}$ because of rule (CONV).

3.3 Specializing Type-Indexed Values

This section is concerned with the specialization of type-indexed values to concrete instances of data types. We have seen in Section 2.1 that the structure of each instance of $\mathit{map}\{\!|\mathsf{t}|\!\}$ rigidly follows the structure of t. Perhaps surprisingly, the intimate correspondence between the type and the value level holds not only for *map* but for all type-indexed values. In fact, the process of specialization can be phrased as an interpretation of the simply typed lambda calculus. The generic programmer specifies the interpretation of type constants. Given this information the meaning of a type term—that is, the specialization of a type-indexed value—is fixed: roughly speaking, type application is interpreted by value application, type abstraction by value abstraction, and type recursion by value recursion.

Before we discuss the formal definitions let us take a look at an example first. Consider specializing *map* for the type Matrix given by $\Lambda \mathsf{a} \,.\, \mathsf{List}\ (\mathsf{List}\ \mathsf{a})$. The instance $\mathit{map}_{\mathsf{Matrix}}$ is given by

$$\begin{aligned}
&\mathit{map}_{\mathsf{Matrix}} :: \forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, (\mathsf{a}_1 \to \mathsf{a}_2) \to (\mathsf{Matrix}\ \mathsf{a}_1 \to \mathsf{Matrix}\ \mathsf{a}_2)\\
&\mathit{map}_{\mathsf{Matrix}} = \lambda \mathsf{a}_1\ \mathsf{a}_2 .\ \lambda \mathit{map}_{\mathsf{a}} :: (\mathsf{a}_1 \to \mathsf{a}_2) \,.\, \mathit{map}_{\mathsf{List}}\ (\mathsf{List}\ \mathsf{a}_1)\ (\mathsf{List}\ \mathsf{a}_2)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{map}_{\mathsf{List}}\ \mathsf{a}_1\ \mathsf{a}_2\ \mathit{map}_{\mathsf{a}}).
\end{aligned}$$

The specialization of the type application $\mathsf{t} = \mathsf{List}\ \mathsf{a}$ is given by the lambda term $t = map_{\mathsf{List}}\ \mathsf{a}_1\ \mathsf{a}_2\ map_{\mathsf{a}}$, which is a combination of universal application and value application. Likewise, the specialization of the application $\mathsf{List}\ \mathsf{t}$ is given by $map_{\mathsf{List}}\ (\mathsf{List}\ \mathsf{a}_1)\ (\mathsf{List}\ \mathsf{a}_2)\ t$. Consequently, if we aim at phrasing the specialization of *map* as a model of the simply typed lambda calculus we must administer both the actual instance of *map* and its type. This observation suggests to represent instances as triples $(\mathsf{s}_1, \mathsf{s}_2; map_{\mathsf{s}})$ where s_1 and s_2 are type schemes of some kind, say, κ and map_{s} is a value term of type $\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2$. Of course, we have to work with equivalence classes of type schemes and terms. Let $\mathcal{E}$ be a set of equations specifying identities between type schemes and/or between terms. Possible identities include $outl\ (t, u) = t$ and $outr\ (t, u) = u$. The applicative structure $\mathcal{M} = (\mathbf{M}, \mathbf{app}, \mathbf{const})$ is then given by

$$\begin{aligned}
\mathbf{M}^{\kappa} &= ([\mathsf{s}_1], [\mathsf{s}_2] \in \mathsf{Scheme}^{\kappa}/\mathcal{E};\ Term^{\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2}/\mathcal{E}) \\
\mathbf{app}_{\kappa,\nu}\ &([\mathsf{r}_1], [\mathsf{r}_2]; [t])\ ([\mathsf{s}_1], [\mathsf{s}_2]; [u]) \\
&= ([\mathsf{r}_1\ \mathsf{s}_1], [\mathsf{r}_2\ \mathsf{s}_2]; [t\ \mathsf{s}_1\ \mathsf{s}_2\ u]) \\
\mathbf{const}(\mathsf{C}) &= ([\mathsf{C}], [\mathsf{C}]; [map\{\![\mathsf{C}]\!\}]).
\end{aligned}$$

Note that the semantic application function **app** uses both the type and the value component of its second argument. It is instructive to check that the resulting term $t\ \mathsf{s}_1\ \mathsf{s}_2\ u$ is indeed well-typed: t has type $\forall \mathsf{a}_1\ \mathsf{a}_2 \,.\, \mathsf{Map}\{\![\kappa]\!\}\ \mathsf{a}_1\ \mathsf{a}_2 \to \mathsf{Map}\{\![\nu]\!\}\ (\mathsf{r}_1\ \mathsf{a}_1)\ (\mathsf{r}_2\ \mathsf{a}_2)$, s_1 and s_2 have kind κ, and u has type $\mathsf{Map}\{\![\kappa]\!\}\ \mathsf{s}_1\ \mathsf{s}_2$. It is important to note that the definitions of $\mathsf{Map}\{\![\kappa \to \nu]\!\}$ and **app** go hand in hand. This explains, in particular, why the definition of $\mathsf{Poly}\{\![\kappa \to \nu]\!\}$ is fixed for function kinds.

Now, does $\mathcal{M}$ also constitute a model? To this end we have to show that $\mathcal{M}$ is extensional and that it satisfies the combinatorial model condition. The first condition is easy to check. To establish the second condition we define combinators (omitting type and kind annotations)

$$\begin{aligned}
\mathbf{K}_{\kappa,\nu} &= ([\mathsf{K}_{\kappa,\nu}], [\mathsf{K}_{\kappa,\nu}]; [\lambda \mathsf{a}_1\ \mathsf{a}_2 \,.\, \lambda map_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2 \,.\, \lambda map_{\mathsf{b}} \,.\, map_{\mathsf{a}}]) \\
\mathbf{S}_{\kappa,\nu,\mu} &= ([\mathsf{S}_{\kappa,\nu,\mu}], [\mathsf{S}_{\kappa,\nu,\mu}]; \\
&\qquad [\lambda \mathsf{a}_1\ \mathsf{a}_2 \,.\, \lambda map_{\mathsf{a}} \,.\, \lambda \mathsf{b}_1\ \mathsf{b}_2 \,.\, \lambda map_{\mathsf{b}} \,.\, \lambda \mathsf{c}_1\ \mathsf{c}_2 \,.\, \lambda map_{\mathsf{c}} \,. \\
&\qquad\quad (map_{\mathsf{a}}\ \mathsf{c}_1\ \mathsf{c}_2\ map_{\mathsf{c}})\ (\mathsf{b}_1\ \mathsf{c}_1)\ (\mathsf{b}_2\ \mathsf{c}_2)\ (map_{\mathsf{b}}\ \mathsf{c}_1\ \mathsf{c}_2\ map_{\mathsf{c}})])
\end{aligned}$$

where K and S are given by

$$\begin{aligned}
\mathsf{K}_{\kappa,\nu} &= \Lambda \mathsf{a} :: \kappa \,.\, \Lambda \mathsf{b} :: \nu \,.\, \mathsf{a} \\
\mathsf{S}_{\kappa,\nu,\mu} &= \Lambda \mathsf{a} :: (\kappa \to \nu \to \mu) \,.\, \Lambda \mathsf{b} :: (\kappa \to \nu) \,.\, \Lambda \mathsf{c} :: \kappa \,.\, (\mathsf{a}\ \mathsf{c})\ (\mathsf{b}\ \mathsf{c}).
\end{aligned}$$

It is straightforward to prove that the combinatory laws are indeed satisfied.

It remains to provide interpretations for the fixed point operators Fix_{κ}. The definition is essentially the same for all type-indexed values. This is why the generic programmer need not supply instances for Fix_{κ} by hand. Here is the definition of $map\{\![\mathsf{Fix}_{\kappa}]\!\}$.

$$\begin{aligned}
map\{\![\mathsf{Fix}_{\kappa}]\!\} = \lambda \mathsf{f}_1\ \mathsf{f}_2 \,.\, \lambda map_{\mathsf{f}} :: (\mathsf{Map}\{\![\kappa \to \kappa]\!\}\ \mathsf{f}_1\ \mathsf{f}_2) \,. \\
fix\ (map_{\mathsf{f}}\ (\mathsf{Fix}_{\kappa}\ \mathsf{f}_1)\ (\mathsf{Fix}_{\kappa}\ \mathsf{f}_2))
\end{aligned}$$

Note that $map\{\!|\mathsf{Fix}_\kappa|\!\}$ essentially equals fix—if we ignore type abstractions and type applications. Let us check that the definition of $map\{\!|\mathsf{Fix}_\kappa|\!\}$ is well-typed. The universal application map_{f} $(\mathsf{Fix}_\kappa\ \mathsf{f}_1)$ $(\mathsf{Fix}_\kappa\ \mathsf{f}_2)$ has type

$$\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2) \to \mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{f}_1\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1))\ (\mathsf{f}_2\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)).$$

Since we have $\mathsf{Fix}_\kappa\ \mathsf{f}_i = \mathsf{f}_i\ (\mathsf{Fix}_\kappa\ \mathsf{f}_i)$, we can use rule (CONV) to infer the type $\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2) \to \mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)$. Consequently, $fix\ (map_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2))$ has type $\mathsf{Map}\{\![\kappa]\!\}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ (\mathsf{Fix}_\kappa\ \mathsf{f}_2)$ as desired.

Now, let us turn to the general case of generic functions. The definitions for arbitrary type-indexed values are very similar to the ones for map. The applicative structure $\mathcal{P} = (\mathbf{P}, \mathbf{app}, \mathbf{const})$ induced by the type-indexed value $poly\{\!|\mathsf{t} :: \kappa|\!\} :: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$ is given by

$$\begin{aligned}
&\mathbf{P}^\kappa = ([\mathsf{s}_1], \ldots, [\mathsf{s}_n] \in \mathsf{Scheme}^\kappa/\mathcal{E};\ \mathit{Term}^{\mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{s}_1\ \ldots\ \mathsf{s}_n}/\mathcal{E})\\
&\mathbf{app}_{\kappa,\nu}\ ([\mathsf{r}_1], \ldots, [\mathsf{r}_n]; [t])\ ([\mathsf{s}_1], \ldots, [\mathsf{s}_n]; [u])\\
&\qquad = ([\mathsf{r}_1\ \mathsf{s}_1], \ldots, [\mathsf{r}_n\ \mathsf{s}_n]; [t\ \mathsf{s}_1\ \ldots\ \mathsf{s}_n\ u])\\
&\mathbf{const}(\mathsf{c}) = ([\mathsf{c}], \ldots, [\mathsf{c}]; [poly\{\!|\mathsf{c}|\!\}]).
\end{aligned}$$

where $poly\{\!|\mathsf{Fix}_\kappa|\!\}$ is defined

$$\begin{aligned}
poly\{\!|\mathsf{Fix}_\kappa|\!\} = \lambda \mathsf{f}_1\ \ldots\ \mathsf{f}_n\,.\,\lambda poly_{\mathsf{f}} :: (\mathsf{Poly}\{\![\kappa \to \kappa]\!\}\ \mathsf{f}_1\ \ldots\ \mathsf{f}_n)\,.\\
fix\ (poly_{\mathsf{f}}\ (\mathsf{Fix}_\kappa\ \mathsf{f}_1)\ \ldots\ (\mathsf{Fix}_\kappa\ \mathsf{f}_n)).
\end{aligned}$$

Three remarks are in order. First, the value domain $\mathbf{P}^\kappa$ is a so-called dependent product: the type of the last component depends on the first n components. A similar structure has also been used to give a semantics to Standard ML's module system, see [35]. Second, if t is a closed type term, then $\mathcal{P}[\![\emptyset \vdash \mathsf{t}::\kappa]\!]\eta$ is of the form $([\mathsf{t}], \ldots, [\mathsf{t}]; [poly\{\!|\mathsf{t}|\!\}])$ where $poly\{\!|\mathsf{t}|\!\}$ is the desired instance. As an aside, note that this is in agreement with $poly$'s type signature $poly\{\!|\mathsf{t} :: \kappa|\!\} :: \mathsf{Poly}\{\![\kappa]\!\}\ \mathsf{t}\ \ldots\ \mathsf{t}$. Third, a type-indexed value can be specialized to a type but not to a type scheme. This restriction is, however, quite mild. Haskell, for instance, does not allow universal quantifiers in **data** declarations. (Recall that we need type schemes to be able to assign types to instances of generic functions.)

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, for instance,

$$\begin{aligned}
poly\{\!|\Lambda \mathsf{a}\,.\,\mathsf{f}\ (\mathsf{g}\ \mathsf{a})|\!\} = \lambda \mathsf{a}_1\ \ldots\ \mathsf{a}_n\,.\,\lambda poly_{\mathsf{a}}\,.\\
poly\{\!|\mathsf{f}|\!\}\ (\mathsf{g}\ \mathsf{a}_1)\ \ldots\ (\mathsf{g}\ \mathsf{a}_n)\ (poly\{\!|\mathsf{g}|\!\}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_n\ poly_{\mathsf{a}}).
\end{aligned}$$

Writing type and function composition as usual this implies, in particular, that $map\{\!|\mathsf{f} \cdot \mathsf{g}|\!\} = map\{\!|\mathsf{f}|\!\} \cdot map\{\!|\mathsf{g}|\!\}$. Perhaps surprisingly, this relationship holds for all type-indexed values, not only for mapping functions. A similar observation is that $poly\{\!|\Lambda \mathsf{a}\,.\,\mathsf{a}|\!\} = \lambda \mathsf{a}\,.\,\lambda poly_{\mathsf{a}}\,.\,poly_{\mathsf{a}}$ for all type-indexed values. Abbreviating $\Lambda \mathsf{a}\,.\,\mathsf{a}$ by Id we have, in particular, that $map\{\!|\mathsf{Id}|\!\} = id$. As an aside, note that these generic identities are not to be confused with the familiar functorial laws $map\{\!|\mathsf{f}|\!\}\ id = id$ and $map\{\!|\mathsf{f}|\!\}\ (\varphi \cdot \psi) = map\{\!|\mathsf{f}|\!\}\ \varphi \cdot map\{\!|\mathsf{f}|\!\}\ \psi$, which are base-level identities.

3.4 Bridging the Gap

The polymorphic lambda 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 $\mathsf{List\ a} = \mathsf{Unit} :+: \mathsf{a} :*: \mathsf{List\ a}$ 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, [45], the Haskell B. Compiler, HBC, [3] and the Haskell interpreter Hugs [29] provide rank-2 type signatures. The latest version of the Glasgow Haskell Compiler, GHC 5.04, even supports general rank-n types. There is, however, one fundamental difference between Haskell and (our presentation) of the polymorphic lambda calculus: Haskell's notion of type equivalence is based on *name equivalence* while the polymorphic lambda calculus employs *structural equivalence*. In the sequel we explain the difference between the two views and show how to adapt the specialization to a type system that is based on name equivalence.

Generic Representation Types. Consider again the Haskell data type of parametric lists:

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

We have modelled this declaration (see Section 3.1) by the type term

$$\mathsf{Fix}\ (\Lambda \mathsf{List}\,.\,\Lambda \mathsf{a}\,.\,\mathsf{Unit} :+: \mathsf{a} :*: \mathsf{List\ a}).$$

Since the equivalence of type terms is based on structural equivalence, we have, in particular, that $\mathsf{List\ a} = \mathsf{Unit} :+: \mathsf{a} :*: \mathsf{List\ a}$ (using (T-FIX), see Figure 3). The specialization of generic values described in the previous section makes essential use of this fact: the List instance of *poly* given by (omitting type abstractions and type applications)

$$\mathit{fix}\ (\lambda \mathit{poly}_{\mathsf{List}}\,.\,\lambda \mathit{poly}_{\mathsf{a}}\,.\,\mathit{poly}_{:+:}\ \mathit{poly}_{\mathsf{Unit}}\ (\mathit{poly}_{:*:}\ \mathit{poly}_{\mathsf{a}}\ (\mathit{poly}_{\mathsf{List}}\ \mathit{poly}_{\mathsf{a}})))$$

only works under the proviso that $\mathsf{List\ a} = \mathsf{Unit} :+: \mathsf{a} :*: \mathsf{List\ a}$. Alas, in Haskell $\mathsf{List\ a}$ is not equivalent to $\mathsf{Unit} :+: \mathsf{a} :*: \mathsf{List\ a}$ as each **data** declaration introduces a new distinct type. Even the type Liste defined by

$$\mathbf{data}\ \mathsf{Liste\ a} = \mathit{Vide} \mid \mathit{Constructeur}\ \mathsf{a}\ (\mathsf{Liste\ a})$$

is not equivalent to List though only the names of the data constructors are different. 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. Fortunately, this can be done 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

$$\textbf{type}\ \mathsf{List}^{\circ}\ \mathsf{a} = \mathsf{Unit}\ {:}{+}{:}\ \mathsf{a}\ {:}{*}{:}\ \mathsf{List}\ \mathsf{a}.$$

As to be expected our generic representation type constructors are just unit, binary sum and binary product. In particular, there is no recursion operator. 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.

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:

$$\begin{array}{ll}
\mathit{from}_{\mathsf{List}} & :: \forall \mathsf{a}\,.\,\mathsf{List}\ \mathsf{a} \to \mathsf{List}^{\circ}\ \mathsf{a} \\
\mathit{from}_{\mathsf{List}}\ \mathit{Nil} & = \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{\mathsf{List}}\ (\mathit{Cons}\ x\ \mathit{xs}) & = \mathit{Inr}\ (x\ {:}{*}{:}\ \mathit{xs}) \\
\mathit{to}_{\mathsf{List}} & :: \forall \mathsf{a}\,.\,\mathsf{List}^{\circ}\ \mathsf{a} \to \mathsf{List}\ \mathsf{a} \\
\mathit{to}_{\mathsf{List}}\ (\mathit{Inl}\ \mathit{Unit}) & = \mathit{Nil} \\
\mathit{to}_{\mathsf{List}}\ (\mathit{Inr}\ (x\ {:}{*}{:}\ \mathit{xs})) & = \mathit{Cons}\ x\ \mathit{xs}.
\end{array}$$

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 *binary* 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

$$\textbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1} \mid \cdots \mid K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}$$

we generate:

$$\textbf{type}\ \mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = \Sigma\ (\Pi\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1})\ \cdots\ (\Pi\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n})$$

where 'Σ' and 'Π' are defined

$$\Sigma\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \begin{cases} \mathsf{t}_1 & \text{if } n = 1 \\ \mathsf{t}_1\ {:}{+}{:}\ \Sigma\ \mathsf{t}_2\ \ldots\ \mathsf{t}_n & \text{if } n > 1 \end{cases}$$

$$\Pi\ \mathsf{t}_1\ \ldots\ \mathsf{t}_n = \begin{cases} \mathsf{Unit} & \text{if } n = 0 \\ \mathsf{t}_1 & \text{if } n = 1 \\ \mathsf{t}_1\ {:}{*}{:}\ \Pi\ \mathsf{t}_2\ \ldots\ \mathsf{t}_n & \text{if } n > 1. \end{cases}$$

Note that this encoding corresponds closely to the scheme introduced in Section 3.1 except that here the argument types of the constructors are *not* recursively encoded. The conversion functions $\mathit{from}_{\mathsf{B}}$ and to_{B} are then given by

$$\begin{array}{ll}
\mathit{from}_{\mathsf{B}} & :: \forall \mathsf{a}_1 \ldots \mathsf{a}_m . \mathsf{B}\ \mathsf{a}_1 \ldots \mathsf{a}_m \to \mathsf{B}^\circ\ \mathsf{a}_1 \ldots \mathsf{a}_m \\
\mathit{from}_{\mathsf{B}}\ (K_1\ x_{11} \ldots x_{1m_1}) & = \mathit{in}^n_1\ (\mathit{tuple}\ x_{11} \ldots x_{1m_1}) \\
\ldots & \\
\mathit{from}_{\mathsf{B}}\ (K_n\ x_{n1} \ldots x_{nm_n}) & = \mathit{in}^n_n\ (\mathit{tuple}\ x_{n1} \ldots x_{nm_n}) \\
\mathit{to}_{\mathsf{B}} & :: \forall \mathsf{a}_1 \ldots \mathsf{a}_m . \mathsf{B}^\circ\ \mathsf{a}_1 \ldots \mathsf{a}_m \to \mathsf{B}\ \mathsf{a}_1 \ldots \mathsf{a}_m \\
\mathit{to}_{\mathsf{B}}\ (\mathit{in}^n_1\ (\mathit{tuple}\ x_{11} \ldots x_{1m_1})) & = K_1\ x_{11} \ldots x_{1m_1} \\
\ldots & \\
\mathit{to}_{\mathsf{B}}\ (\mathit{in}^n_n\ (\mathit{tuple}\ x_{n1} \ldots x_{nm_n})) & = K_n\ x_{n1} \ldots x_{nm_n}
\end{array}$$

where

$$\mathit{in}^n_i\ t = \begin{cases} t & \text{if } n = 1 \\ \mathit{Inl}\ t & \text{if } n > 1 \wedge i = 1 \\ \mathit{Inr}\ (\mathit{in}^{n-1}_{i-1}\ t) & \text{if } n > 1 \wedge i > 1 \end{cases}$$

$$\mathit{tuple}\ t_1 \ldots t_n = \begin{cases} \mathit{Unit} & \text{if } n = 0 \\ t_1 & \text{if } n = 1 \\ (t_1 :\!*\!: \mathit{tuple}\ t_2 \ldots t_n) & \text{if } n > 1. \end{cases}$$

An alternative encoding, which is based on a binary sub-division scheme, is given in [18]. 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).

Specializing Generic Values. Assume for the sake of example that we want to specialize the generic functions *encode* and *decodes* introduced in Section 1.3 to the List data type. Recall the types of the generic values (here expressed as type synonyms):

$$\begin{array}{l}
\mathbf{type}\ \mathsf{Encode\ a} \ = \mathsf{a} \to \mathsf{Bin} \\
\mathbf{type}\ \mathsf{Decodes\ a} = \mathsf{Bin} \to (\mathsf{a}, \mathsf{Bin}).
\end{array}$$

Since List° involves only the type constants Unit, ':+:' and ':*:' (and the type variables List and a), we can easily specialize *encode* and *decodes* to List° a: the instances have types Encode (List° a) and Decodes (List° a), respectively. However, we want to generate functions of type Encode (List a) and Decodes (List a). Now, we already know how to convert between List° a and List a. So it remains to lift $\mathit{from}_{\mathsf{List}}$ and $\mathit{to}_{\mathsf{List}}$ to functions of type Encode (List a) $\to$ Encode (List° a) and Encode (List° a) $\to$ Encode (List a). But this lifting is exactly what a mapping function does! In particular, since Encode and Decodes involve function types and we have to convert to and fro, we can use the embedding-projection maps of Section 2.3 for this purpose.

For *mapE* we have to package the two conversion functions into a single value:

$$\begin{array}{l}
\mathit{conv}_{\mathsf{List}} :: \forall \mathsf{a} . \mathsf{EP}\ (\mathsf{List\ a})\ (\mathsf{List}^\circ\ \mathsf{a}) \\
\mathit{conv}_{\mathsf{List}} = \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{List}}, \mathit{to} = \mathit{to}_{\mathsf{List}}\}.
\end{array}$$

Then $\mathit{encode}_{\mathsf{List}}$ and $\mathit{decodes}_{\mathsf{List}}$ are given by

$$\begin{aligned} \mathit{encode}_{\mathsf{List}}\ \mathit{ena} &= \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Encode}|\!\}\ \mathit{conv}_{\mathsf{List}})\ (\mathit{encode}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{ena}) \\ \mathit{decodes}_{\mathsf{List}}\ \mathit{dea} &= \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Decodes}|\!\}\ \mathit{conv}_{\mathsf{List}})\ (\mathit{decodes}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{dea}). \end{aligned}$$

Consider the definition of $\mathit{encode}_{\mathsf{List}}$. The specialization $\mathit{encode}\{\!|\mathsf{List}^{\circ}|\!\}\ \mathit{ena}$ yields a function of type $\mathsf{Encode}\ (\mathsf{List}^{\circ}\ \mathsf{a})$; the call $\mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Encode}|\!\}\ \mathit{conv}_{\mathsf{List}})$ then converts this function into a value of type $\mathsf{Encode}\ (\mathsf{List}\ \mathsf{a})$ as desired.

In general, the translation proceeds as follows. For each generic definition we generate the following.

– A type synonym $\mathsf{Poly} = \mathsf{Poly}\{[\star]\}$ for the type of the generic value.
– An embedding-projection map, $\mathit{mapE}\{\!|\mathsf{Poly}|\!\}$.
– Generic instances for Unit, ‘:+:’, ‘:*:’ 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.
– An embedding-projection pair $\mathit{conv}_{\mathsf{B}}$ that converts between $\mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m$ and $\mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m$.

$$\begin{aligned} \mathit{conv}_{\mathsf{B}} &:: \forall \mathsf{a}_1 \ldots \mathsf{a}_m\,.\,\mathsf{EP}\ (\mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m)\ (\mathsf{B}^{\circ}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m) \\ \mathit{conv}_{\mathsf{B}} &= \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{B}}, \mathit{to} = \mathit{to}_{\mathsf{B}}\} \end{aligned}$$

The instance of poly for type $\mathsf{B} :: \kappa$ is then given by (using Haskell syntax)

$$\begin{aligned} &\mathit{poly}_{\mathsf{B}} :: \mathsf{Poly}\{[\kappa]\}\ \mathsf{B}\ \ldots\ \mathsf{B} \\ &\mathit{poly}_{\mathsf{B}}\ \mathit{poly}_{\mathsf{a}_1}\ \ldots\ \mathit{poly}_{\mathsf{a}_m} \\ &\quad = \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{Poly}|\!\}\ \mathit{conv}_{\mathsf{B}}\ \ldots\ \mathit{conv}_{\mathsf{B}})\ (\mathit{poly}\{\!|\mathsf{B}^{\circ}|\!\}\ \mathit{poly}_{\mathsf{a}_1}\ \ldots\ \mathit{poly}_{\mathsf{a}_m}). \end{aligned}$$

If $\mathsf{Poly}\{[\kappa]\}\ \mathsf{B}\ \ldots\ \mathsf{B}$ has a rank of 2 or below, we can express $\mathit{poly}_{\mathsf{B}}$ directly in Haskell. Figures 7 and 8 show several examples of specializations all expressed in Haskell.

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:

$$\begin{aligned} &\mathit{mapE}\{\!|\mathsf{t} :: \kappa|\!\} && :: \mathsf{MapE}\{[\kappa]\}\ \mathsf{t}\ \mathsf{t} \\ &\mathit{mapE}\{\!|\mathsf{Char}|\!\} && = \mathit{id}_E \\ &\mathit{mapE}\{\!|\mathsf{Int}|\!\} && = \mathit{id}_E \\ &\mathit{mapE}\{\!|\rightarrow|\!\}\ \mathit{mapE}_{\mathsf{a}}\ \mathit{mapE}_{\mathsf{b}} && = \mathit{mapE}_{\mathsf{a}} \rightarrow_E \mathit{mapE}_{\mathsf{b}} \\ &\mathit{mapE}\{\!|\mathsf{IO}|\!\}\ \mathit{mapE}_{\mathsf{a}} && = \mathit{EP}\{\mathit{from} = \mathit{fmap}\ (\mathit{from}\ \mathit{mapE}_{\mathsf{a}}), \\ & && \qquad\quad \mathit{to} = \mathit{fmap}\ (\mathit{to}\ \mathit{mapE}_{\mathsf{a}})\}. \end{aligned}$$

Note that in the last equation *mapE* falls back on the ‘ordinary’ mapping function *fmap*. (In Haskell, *fmap*, a method of the $\mathsf{Functor}$ class, is an overloaded version of *map*, which is confined to lists.) In fact, we can alternatively define

$$\mathit{mapE}\{\!|\mathsf{IO}|\!\} = \mathit{liftE}$$

where

```
 {- binary encoding -}
type Encode a                 = a → Bin
encode_Unit                   :: Encode Unit
encode_Unit                   = λUnit → []
encode_:+:                    :: ∀a . Encode a → (∀b . Encode b → Encode (a :+: b))
encode_:+: encode_a encode_b  = λs → case s of { Inl a → 0 : encode_a a;
                                                 Inr b → 1 : encode_b b}
encode_:*:                    :: ∀a . Encode a → (∀b . Encode b → Encode (a :*: b))
encode_:*: encode_a encode_b  = λ(a :*: b) → encode_a a ++ encode_b b
mapE_Encode                   :: ∀a b . EP a b → EP (Encode a) (Encode b)
mapE_Encode m                 = EP{from = λh → h · to m, to = λh → h · from m}
 {- equality -}
type Equal a_1 a_2            = a_1 → a_2 → Bool
equal_Unit                    :: Equal Unit Unit
equal_Unit                    = λUnit Unit → True
equal_:+:                     :: ∀a_1 a_2 . Equal a_1 a_2 → (∀b_1 b_2 . Equal b_1 b_2
                                       → Equal (a_1 :+: b_1) (a_2 :+: b_2))
equal_:+: equal_a equal_b     = λs_1 s_2 → case (s_1, s_2) of
                                    {(Inl a_1, Inl a_2) → equal_a a_1 a_2;
                                     (Inl a_1, Inr b_2) → False;
                                     (Inr b_1, Inl a_2) → False;
                                     (Inr b_1, Inr b_2) → equal_b b_1 b_2}
equal_:*:                     :: ∀a_1 a_2 . Equal a_1 a_2 → (∀b_1 b_2 . Equal b_1 b_2
                                       → Equal (a_1 :*: b_1) (a_2 :*: b_2))
equal_:*: equal_a equal_b     = λ(a_1 :*: b_1) (a_2 :*: b_2) → equal_a a_1 a_2 ∧ equal_b b_1 b_2
mapE_Equal                    :: ∀a_1 b_1 . EP a_1 b_1 → (∀a_2 b_2 . EP a_2 b_2
                                       → EP (Equal a_1 a_2) (Equal b_1 b_2))
mapE_Equal m_1 m_2            = EP{from = λh → λa_1 a_2 → h (to m_1 a_1) (to m_2 a_2),
                                   to = λh → λb_1 b_2 → h (from m_1 b_1) (from m_2 b_2)}
 {- generic representation types -}
type Maybe° a                 = Unit :+: a
from_Maybe                    :: ∀a . Maybe a → Maybe° a
from_Maybe Nothing            = Inl Unit
from_Maybe (Just a)           = Inr a
to_Maybe                      :: ∀a . Maybe° a → Maybe a
to_Maybe (Inl Unit)           = Nothing
to_Maybe (Inr a)              = Just a
conv_Maybe                    :: ∀a . EP (Maybe a) (Maybe° a)
conv_Maybe                    = EP{from = from_Maybe, to = to_Maybe}
```

Fig. 7. Specializing generic values in Haskell (part 1).

```
type List° a                = Unit :+: a :*: List a
from_List                   :: ∀a . List a → List° a
from_List []                = Inl Unit
from_List (a : as)          = Inr (a :*: as)
to_List                     :: ∀a . List° a → List a
to_List (Inl Unit)          = []
to_List (Inr (a :*: as))    = a : as
conv_List                   :: ∀a . EP (List a) (List° a)
conv_List                   = EP{from = from_List, to = to_List}
type GRose° f a             = a :*: f (GRose f a)
from_GRose                  :: ∀f a . GRose f a → GRose° f a
from_GRose (GBranch a ts)   = (a :*: ts)
to_GRose                    :: ∀f a . GRose° f a → GRose f a
to_GRose (a :*: ts)         = GBranch a ts
conv_GRose                  :: ∀f a . EP (GRose f a) (GRose° f a)
conv_GRose                  = EP{from = from_GRose, to = to_GRose}

  {- specializing binary encoding -}
encode_Maybe          :: ∀a . Encode a → Encode (Maybe a)
encode_Maybe encode_a = to (mapE_Encode conv_Maybe) (encode_:+: encode_Unit encode_a)
encode_List           :: ∀a . Encode a → Encode (List a)
encode_List encode_a  = to (mapE_Encode conv_List) (
                            encode_:+: encode_Unit (encode_:*: encode_a (encode_List encode_a)))
encode_GRose          :: ∀f . (∀b . Encode b → Encode (f b))
                                → (∀a . Encode a → Encode (GRose f a))
encode_GRose encode_f encode_a
                      = to (mapE_Encode conv_GRose) (
                            encode_:*: encode_a (encode_f (encode_GRose encode_f encode_a)))
  {- specializing equality -}
equal_Maybe           :: ∀a1 a2 . Equal a1 a2 → Equal (Maybe a1) (Maybe a2)
equal_Maybe equal_a   = to (mapE_Equal conv_Maybe conv_Maybe) (equal_:+: equal_Unit equal_a)
equal_List            :: ∀a1 a2 . Equal a1 a2 → Equal (List a1) (List a2)
equal_List equal_a    = to (mapE_Equal conv_List conv_List) (
                            equal_:+: equal_Unit (equal_:*: equal_a (equal_List equal_a)))
equal_GRose           :: ∀f1 f2 . (∀b1 b2 . Equal b1 b2 → Equal (f1 b1) (f2 b2))
                                   → (∀a1 a2 . Equal a1 a2
                                                → Equal (GRose f1 a1) (GRose f2 a2))
equal_GRose equal_f equal_a
                      = to (mapE_Equal conv_GRose conv_GRose) (
                            equal_:*: equal_a (equal_f (equal_GRose equal_f equal_a)))
```

Fig. 8. Specializing generic values in Haskell (part 2).

$$\begin{aligned}&\mathit{liftE} &&:: \forall \mathsf{f}\,.\,(\mathsf{Functor}\ \mathsf{f}) \Rightarrow \forall \mathsf{a}\ \mathsf{a}^{\circ}\,.\,\mathsf{EP}\ \mathsf{a}\ \mathsf{a}^{\circ} \to \mathsf{EP}\ (\mathsf{f}\ \mathsf{a})\ (\mathsf{f}\ \mathsf{a}^{\circ})\\ &\mathit{liftE}\ \mathit{mapE}_{\mathsf{a}} &&= \mathit{EP}\{\mathit{from} = \mathit{fmap}\ (\mathit{from}\ \mathit{mapE}_{\mathsf{a}}), \mathit{to} = \mathit{fmap}\ (\mathit{to}\ \mathit{mapE}_{\mathsf{a}})\}.\end{aligned}$$

Now, the $\mathsf{Poly} :: \pi$ instance of *mapE* is given by

$$\begin{aligned}&\mathit{mapE}_{\mathsf{Poly}} &&:: \mathsf{MapE}\{[\pi]\}\ \mathsf{Poly}\ \mathsf{Poly}\\ &\mathit{mapE}_{\mathsf{Poly}}\ \mathit{mapE}_{\mathsf{a}_1}\ \ldots\ \mathit{mapE}_{\mathsf{a}_k} &&= \mathit{mapE}\{\!|\mathsf{Poly}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_k|\!\}\ \rho.\end{aligned}$$

where $\rho = (\mathsf{a}_1 := \mathit{mapE}_{\mathsf{a}_1}, \ldots, \mathsf{a}_k := \mathit{mapE}_{\mathsf{a}_k})$ is an environment mapping type variables to terms. We use for the first time an explicit environment in order to be able to extend the definition to polymorphic types. Recall that the specialization of generic values does not work for polymorphic types. However, we allow polymorphic types to occur in the type signature of a generic value. Fortunately, we can extend *mapE* so that it works for universal quantification over types of kind $\star$ and kind $\star \to \star$.

$$\begin{aligned}&\mathit{mapE}\{\!|\mathsf{C}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{C}|\!\}\\ &\mathit{mapE}\{\!|\mathsf{a}|\!\}\ \rho &&= \rho(\mathsf{a})\\ &\mathit{mapE}\{\!|\mathsf{t}\ \mathsf{u}|\!\}\ \rho &&= (\mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho)\ (\mathit{mapE}\{\!|\mathsf{u}|\!\}\ \rho)\\ &\mathit{mapE}\{\!|\forall \mathsf{a} :: \star\,.\,\mathsf{t}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho(\mathsf{a} := \mathit{id}_E)\\ &\mathit{mapE}\{\!|\forall \mathsf{f} :: \star \to \star\,.\,(\mathsf{Functor}\ \mathsf{f}) \Rightarrow \mathsf{t}|\!\}\ \rho &&= \mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho(\mathsf{f} := \mathit{liftE}).\end{aligned}$$

Two remarks are in order.

Haskell has neither type abstraction 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 $\mathsf{Poly}\ \mathsf{a} = \forall \mathsf{f}\,.\,\mathsf{f}\ \mathsf{a} \to \mathsf{f}\ \mathsf{a}$. There is no mapping function that works uniformly for all f. For that reason we have to restrict f to instances of $\mathsf{Functor}$ 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 id_E.

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. One possibility to break the spell is to implement *mapE* for the B data type 'by hand'. Fortunately *mapE* is very well-behaved, so the code generation is straightforward. The embedding-projection map for the data type $\mathsf{B} :: \kappa$

$$\mathbf{data}\ \mathsf{B}\ \mathsf{a}_1\ \ldots\ \mathsf{a}_m = K_1\ \mathsf{t}_{11}\ \ldots\ \mathsf{t}_{1m_1}\ |\ \cdots\ |\ K_n\ \mathsf{t}_{n1}\ \ldots\ \mathsf{t}_{nm_n}$$

is given by

$$\begin{aligned}&\mathit{mapE}_{\mathsf{B}} &&:: \mathsf{MapE}\{[\kappa]\}\ \mathsf{B}\ \mathsf{B}\\ &\mathit{mapE}_{\mathsf{B}}\ \mathit{mapE}_{\mathsf{a}_1}\ \ldots\ \mathit{mapE}_{\mathsf{a}_m} &&\\ & &&= \mathit{EP}\{\mathit{from} = \mathit{from}_{\mathsf{B}}, \mathit{to} = \mathit{to}_{\mathsf{B}}\}\\ &\quad\mathbf{where} &&\\ &\quad\mathit{from}_{\mathsf{B}}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) &&= K_1\ (\mathit{from}\{\!|\mathsf{t}_{11}|\!\}\ \rho\ x_{11})\ \ldots\ (\mathit{from}\{\!|\mathsf{t}_{1m_1}|\!\}\ \rho\ x_{1m_1})\\ &\quad\ldots &&\\ &\quad\mathit{from}_{\mathsf{B}}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) &&= K_n\ (\mathit{from}\{\!|\mathsf{t}_{n1}|\!\}\ \rho\ x_{n1})\ \ldots\ (\mathit{from}\{\!|\mathsf{t}_{nm_n}|\!\}\ \rho\ x_{nm_n})\\ &\quad\mathit{to}_{\mathsf{B}}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) &&= K_1\ (\mathit{to}\{\!|\mathsf{t}_{11}|\!\}\ \rho\ x_{11})\ \ldots\ (\mathit{to}\{\!|\mathsf{t}_{1m_1}|\!\}\ \rho\ x_{1m_1})\\ &\quad\ldots &&\\ &\quad\mathit{to}_{\mathsf{B}}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) &&= K_n\ (\mathit{to}\{\!|\mathsf{t}_{n1}|\!\}\ \rho\ x_{n1})\ \ldots\ (\mathit{to}\{\!|\mathsf{t}_{nm_n}|\!\}\ \rho\ x_{nm_n})\end{aligned}$$

where $\mathit{from}\{\!|\mathsf{t}|\!\}\ \rho = \mathit{from}\ (\mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho)$, $\mathit{to}\{\!|\mathsf{t}|\!\}\ \rho = \mathit{to}\ (\mathit{mapE}\{\!|\mathsf{t}|\!\}\ \rho)$, and $\rho = (\mathsf{a}_1 := \mathit{mapE}_{\mathsf{a}_1}, \ldots, \mathsf{a}_m := \mathit{mapE}_{\mathsf{a}_m})$. For example, for Encode and Decodes we obtain

$$\begin{aligned}
&\mathit{mapE}_{\mathsf{Encode}} && :: \forall \mathsf{a}\ \mathsf{a}^{\circ}\,.\, \mathsf{EP}\ \mathsf{a}\ \mathsf{a}^{\circ} \to \mathsf{EP}\ (\mathsf{Encode}\ \mathsf{a})\ (\mathsf{Encode}\ \mathsf{a}^{\circ})\\
&\mathit{mapE}_{\mathsf{Encode}}\ \mathit{mapE}_{\mathsf{a}} && = \mathit{mapE}_{\to}\ \mathit{mapE}_{\mathsf{a}}\ \mathit{id}_E\\
&\mathit{mapE}_{\mathsf{Decodes}} && :: \forall \mathsf{a}\ \mathsf{a}^{\circ}\,.\, \mathsf{EP}\ \mathsf{a}\ \mathsf{a}^{\circ} \to \mathsf{EP}\ (\mathsf{Decodes}\ \mathsf{a})\ (\mathsf{Decodes}\ \mathsf{a}^{\circ})\\
&\mathit{mapE}_{\mathsf{Decodes}}\ \mathit{mapE}_{\mathsf{a}} && = \mathit{mapE}_{\to}\ \mathit{id}_E\ (\mathit{mapE}_{(,)}\ \mathit{mapE}_{\mathsf{a}}\ \mathit{id}_E)
\end{aligned}$$

where the mapping function $\mathit{mapE}_{(,)}$ is generated according to the scheme above:

$$\begin{aligned}
&\mathit{mapE}_{(,)} && :: \forall \mathsf{a}\ \mathsf{a}^{\circ}\,.\, \mathsf{EP}\ \mathsf{a}\ \mathsf{a}^{\circ} \to \forall \mathsf{b}\ \mathsf{b}^{\circ}\,.\, \mathsf{EP}\ \mathsf{b}\ \mathsf{b}^{\circ} \to \mathsf{EP}\ (\mathsf{a},\mathsf{b})\ (\mathsf{a}^{\circ},\mathsf{b}^{\circ})\\
&\mathit{mapE}_{(,)}\ \mathit{mapE}_{\mathsf{a}}\ \mathit{mapE}_{\mathsf{b}} && = \mathit{EP}\{\mathit{from} = \mathit{from}_{(,)}, \mathit{to} = \mathit{to}_{(,)}\}\\
&\quad \textbf{where}\ \mathit{from}_{(,)}\ (a,b) && = (\mathit{from}\ \mathit{mapE}_{\mathsf{a}}\ a, \mathit{from}\ \mathit{mapE}_{\mathsf{b}}\ b)\\
&\qquad\qquad \mathit{to}_{(,)}\ (a,b) && = (\mathit{to}\ \mathit{mapE}_{\mathsf{a}}\ a, \mathit{to}\ \mathit{mapE}_{\mathsf{b}}\ b).
\end{aligned}$$

Interestingly, the Generic Haskell compiler does not treat *mapE* in a special way. Rather, it uses the same specialization mechanism also for *mapE* instances. This is possible because *mapE*'s type involves only the type EP, for which we can specialize *mapE* by hand.

4 Conclusion

We have presented Generic Haskell, an extension of Haskell that allows the definition of generic programs. We have shown how to implement typical examples of generic programs such as equality, pretty printing, mapping functions and reductions. The central idea is to define a generic function by induction on the structure of types. Haskell possesses a rich type system, which essentially corresponds to the simply typed lambda calculus (with kinds playing the rôle of types). This type system presents a real challenge: how can we define generic functions and how can we assign types to these functions? It turns out that type-indexed values possess kind-indexed types, types that are defined by induction on the structure of kinds.

Though generic programming adds an extra level of abstraction to programming, it is in many cases simpler than conventional programming. The fundamental reason is that genericity gives you a lot of things for free. For instance, the generic programmer only has to provide cases for primitive types and for binary sums and products. Generic Haskell automatically takes care of type abstraction, type application, and type recursion.

Generic Haskell takes a transformational approach to generic programming: a generic function is translated into a family of polymorphic functions. We have seen that this transformation can be phrased as an interpretation of the simply typed lambda calculus. One of the benefits of this approach—not mentioned in these notes—is that it is possible to adapt one of the main tools for studying typed lambda calculi, logical relations, to generic reasoning, see [22]. To prove a generic property it suffices to prove the assertion for type constants. Everything else is taken care of automatically.

Finally, we have shown how to adapt the technique of specialization to Haskell, whose type system is based on name equivalence.

Acknowledgement

Thanks are due to Andres Löh for implementing the Generic Haskell compiler.

References

1. Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Th. Arts and M. Mohnen, editors, *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 257–278, lvsj, Sweden, September 2001.
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Jeuring Jeuring, editors, *Pre-Proceedings of IFIP TC2 Working Conf. on Generic Programming, WCGP'02, Dagstuhl, 11–12 July 2002*, 2002. (Final Proceedings to be published by Kluwer Acad. Publ.).
3. Lennart Augustsson. *The Haskell B. Compiler (HBC)*, 1998. Available from `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`.
4. Lennart Augustsson. Cayenne – a language with dependent types. *SIGPLAN Notices*, 34(1):239–250, January 1999.
5. Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In Simon Peyton Jones, editor, *Proceedings of the 2002 International Conference on Functional Programming, Pittsburgh, PA, USA, October 4-6, 2002*, pages 157–166. ACM Press, October 2002.
6. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Europe, London, 2nd edition, 1998.
7. Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, June 1998.
8. Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
9. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M.T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, October 2002.
10. Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.
11. Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Aart Middeldorp and Taisuke Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, volume 1722 of *Lecture Notes in Computer Science*, pages 241–250. Springer-Verlag, November 1999.

12. Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed (functional pearl). In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP'00)*, volume 35 of *ACM Sigplan Notices*, pages 221–232, New York, September 2000. ACM Press.
13. G. Gierz, K.H. Hofmann, K Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.
14. Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.
15. T. Hagino. *Category Theoretic Approach to Data Types.* PhD thesis, University of Edinburgh, 1987.
16. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, California*, pages 130–141. ACM Press, 1995.
17. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
18. Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.
19. Ralf Hinze. Polytypic programming with ease (extended abstract). In Aart Middeldorp and Taisuke Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan*, volume 1722 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, November 1999.
20. Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.
21. Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.
22. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programmming*, 43:129–159, 2002.
23. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, August 2001. The preliminary proceedings appeared as a University of Nottingham technical report.
24. Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.
25. Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In S. Doaitse Swierstra, editor, *Proceedings European Symposium on Programming, ESOP'99*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287, Berlin, 1999. Springer-Verlag.
26. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.
27. C.B. Jay and J.R.B. Cocket. Shapely types and shape polymorphism. In D. Sanella, editor, *Programming Languages and Systems — ESOP'94: 5th European Symposium on Programming, Edinburgh, UK, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316, Berlin, 11–13 April 1994. Springer-Verlag.

28. Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd International School on Advanced Functional Programming, Olympia, WA, USA*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.
29. M.P. Jones and J.C. Peterson. *Hugs 98 User Manual*, May 1999. Available from `http://www.haskell.org/hugs`.
30. G. Malcolm. *Algebraic data types and program transformation.* PhD thesis, University of Groningen, 1990.
31. Lambert Meertens. Calculate polytypically! In H. Kuchen and S.D. Swierstra, editors, *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1996.
32. Erik Meijer and Graham Hutton. 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*, pages 324–333. ACM Press, June 1995.
33. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
34. John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, Cambridge, MA, 1996.
35. Eugenio Moggi. A cateogry-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1(1):103–139, March 1991.
36. Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228, 1984.
37. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
38. Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.
39. Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, 22–24 April*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer-Verlag, 1996.
40. Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, Mass., 2002.
41. Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.* Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.
42. Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan, 1992.
43. Tim Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
44. Tim Sheard. 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, November 1993.

45. The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2003. Available from `http://www.haskell.org/ghc/documentation.html`.
46. Philip Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89), London, UK*, pages 347–359. Addison-Wesley Publishing Company, September 1989.
47. Philip Wadler. The Girard-Reynolds isomorphism. In N. Kobayashi and B. C. Pierce, editors, *Proc. of 4th Int. Symp. on Theoretical Aspects of Computer Science, TACS 2001, Sendai, Japan, 29–31 Oct. 2001*, volume 2215 of *Lecture Notes in Computer Science*, pages 468–491. Springer-Verlag, Berlin, 2001.
48. Philip Wadler. A prettier printer. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 223–243. Palgrave Macmillan Publishers Ltd, March 2003.
49. Stephanie Weirich. Higher-order intensional type analysis. In D. Le Métayer, editor, *Proceedings of the 11th European Symposium on Programming, ESOP 2002*, volume 2305 of *Lecture Notes in Computer Science*, pages 98–114, 2002.

Generic Haskell: Applications

Ralf Hinze[1] and Johan Jeuring[2,3]

[1] Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
`ralf@informatik.uni-bonn.de`
`http://www.informatik.uni-bonn.de/~ralf/`
[2] Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
`johanj@cs.uu.nl`
`http://www.cs.uu.nl/~johanj/`
[3] Open University, Heerlen, The Netherlands

Abstract. Generic Haskell is an extension of Haskell that supports the construction of generic programs. These lecture notes discuss three advanced generic programming applications: generic dictionaries, compressing XML documents, and the zipper: a data structure used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down the tree. When describing and implementing these examples, we will encounter some advanced features of Generic Haskell, such as type-indexed data types, dependencies between and generic abstractions of generic functions, adjusting a generic function using a default case, and generic functions with a special case for a particular constructor.

1 Introduction

A generic (or polytypic, type-indexed) function is a function that can be instantiated on many data types to obtain data type specific functionality. Examples of generic functions are the functions that can be derived in Haskell [45], such as *show*, *read*, and '`==`'. In the first part of these lecture notes, entitled Generic Haskell: Practice and Theory [22], we have introduced generic functions, and we have shown how to implement them in Generic Haskell [9]. We assume the reader is familiar with the first part of these notes. For an introduction to a related but different kind of generic programming, see Backhouse et al [2].

Why is generic programming important? Generic programming makes programs easier to write:

– Programs that could only be written in an untyped style can now be written in a language with types.
– Some programs come for free.
– Some programs are simple adjustments of library functions, instead of complicated traversal functions.

Of course not all programs become simpler when you write your programs in a generic programming language, but, on the other hand, no programs become more complicated. In this paper we will try to give you a feeling about where and when generic programs are useful.

These lecture notes describe three advanced generic programming applications: generic dictionaries, compressing XML documents, and the zipper. The applications are described in more detail below. In the examples, we will encounter several new generic programming concepts:

– *Type-indexed data types.* A type-indexed data type is constructed in a generic way from an argument data type [23]. It is the equivalent of a type-indexed function on the level of data types.
– *Default cases.* To define a generic function that is the same as another function except for a few cases we use a default case [10]. If the new definition does not provide a certain case, then the default case applies and copies the case from another function.
– *Constructor cases.* A constructor case of a generic program deals with a constructor of a data type that requires special treatment [10]. Constructor cases are especially useful when dealing with data types with a large number of constructors, and only a small number of constructors need special treatment.
– *Dependencies and generic abstractions.* To write a generic function that uses another generic function we can use a dependency or a generic abstraction [10].

We will introduce these concepts as we go along.

Example 1: Digital Searching. A digital search tree or trie is a search tree scheme that employs the structure of search keys to organize information. Searching is useful for various data types, so we would like to allow for keys and information of any data type. This means that we have to construct a new kind of trie for each key type. For example, consider the data type String defined by

```
data String = Nil | Cons Char String.
```

We can represent string-indexed tries with associated values of type v as follows:

```
data FMap_String v = Null_String
                   | Node_String (Maybe v)
                                 (FMapChar (FMap_String v)).
```

Such a trie for strings would typically be used for an index on texts. The constructor *Null_String* represents the empty trie. The first component of the constructor *Node_String* contains the value associated with *Nil*. The second component of *Node_String* is derived from the constructor *Cons* :: Char → String → String. We assume that a suitable data structure, FMapChar, and an associated lookup function *lookupChar* :: ∀v . Char → FMapChar v → Maybe v for characters are predefined. Given these prerequisites we can define a look-up function for strings as follows:

$$\begin{aligned}
&\mathit{lookup\_String} :: \mathsf{String} \to \mathsf{FMap\_String}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}\\
&\mathit{lookup\_String}\ s\ \mathit{Null\_String} \qquad\qquad = \mathit{Nothing}\\
&\mathit{lookup\_String}\ \mathit{Nil}\ (\mathit{Node\_String}\ \mathit{tn}\ \mathit{tc}) = \mathit{tn}\\
&\mathit{lookup\_String}\ (\mathit{Cons}\ c\ s)\ (\mathit{Node\_String}\ \mathit{tn}\ \mathit{tc})\\
&\quad = (\mathit{lookupChar}\ c \diamond \mathit{lookup\_String}\ s)\ \mathit{tc}.
\end{aligned}$$

To look up a non-empty string, *Cons c s*, we look up *c* in the FMapChar obtaining a trie, which is then recursively searched for *s*. Since the look-up functions have result type Maybe v, we use the reverse monadic composition of the Maybe monad, denoted by '$\diamond$', to compose *lookup_String* and *lookupChar*.

$$\begin{aligned}
&(\diamond) \qquad\quad :: (\mathsf{a} \to \mathsf{Maybe}\ \mathsf{b}) \to (\mathsf{b} \to \mathsf{Maybe}\ \mathsf{c}) \to \mathsf{a} \to \mathsf{Maybe}\ \mathsf{c}\\
&(f \diamond g)\ a = \mathbf{case}\ f\ a\ \mathbf{of}\ \{\mathit{Nothing} \to \mathit{Nothing}; \mathit{Just}\ b \to g\ b\}
\end{aligned}$$

Consider now the data type Bush of binary trees with characters in the leaves:

$$\mathbf{data}\ \mathsf{Bush} = \mathit{Leaf}\ \mathsf{Char} \mid \mathit{Bin}\ \mathsf{Bush}\ \mathsf{Bush}.$$

Bush-indexed tries can be represented by the following data type:

$$\begin{aligned}
\mathbf{data}\ \mathsf{FMap\_Bush}\ \mathsf{v} = {} & \mathit{Null\_Bush}\\
\mid {} & \mathit{Node\_Bush}\ (\mathsf{FMapChar}\ \mathsf{v})\\
& \qquad\qquad (\mathsf{FMap\_Bush}\ (\mathsf{FMap\_Bush}\ \mathsf{v})).
\end{aligned}$$

Again, we have two components, one to store values constructed by *Leaf*, and one for values constructed by *Bin*. The corresponding look-up function is given by

$$\begin{aligned}
&\mathit{lookup\_Bush} :: \mathsf{Bush} \to \mathsf{FMap\_Bush}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}\\
&\mathit{lookup\_Bush}\ b\ \mathit{Null\_Bush} \qquad\qquad\quad = \mathit{Nothing}\\
&\mathit{lookup\_Bush}\ (\mathit{Leaf}\ c)\ (\mathit{Node\_Bush}\ \mathit{tl}\ \mathit{tf}) = \mathit{lookupChar}\ c\ \mathit{tl}\\
&\mathit{lookup\_Bush}\ (\mathit{Bin}\ \mathit{bl}\ \mathit{br})\ (\mathit{Node\_Bush}\ \mathit{tl}\ \mathit{tf})\\
&\quad = (\mathit{lookup\_Bush}\ \mathit{bl} \diamond \mathit{lookup\_Bush}\ \mathit{br})\ \mathit{tf}.
\end{aligned}$$

One can easily recognize that not only the look-up functions, but also the data types for the tries are instances of an underlying generic pattern. In the following section we will show how to define a trie and associated functions generically for arbitrary data types.

Example 2: Compressing XML Documents. The extensible markup language XML [49] is used to mark up text with structure information. XML documents may become (very) large because of the markup that is added to the content. A good XML compressor can compress an XML document by quite a large factor.

An XML document is usually structured according to a DTD (Document Type Definition), a specification that describes which tags may be used in the XML document, and in which positions and order they have to be. A DTD is, in a way, the *type* of an XML document. An XML document is called *valid* with respect to a certain DTD if it follows the structure that is specified by that DTD. An XML compressor can use information from the DTD to obtain better compression. For example, consider the following small XML file:

```
<book lang="English">
<title>  Dead Famous </title>
<author> Ben Elton   </author>
<date>   2001        </date>
</book>.
```

This file may be compressed by separating the structure from the data, and compressing the two parts separately. For compressing the structure we can make good use of the DTD of the document.

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST book lang (English | Dutch) #REQUIRED>
```

If we know how many elements and attributes, say n, appear in the DTD (the DTD above document contains 6 elements), we can replace the markup of an element in an XML file which is valid with respect to the DTD by a natural number between 0 and $n-1$, or by $log_2\ n$ bits. This is one of the main ideas behind XMill [36]. We improve on XMill by only recording the choices made in an XML document. In the above document, there is a choice for the language of the book, and the number of chapters it has. All the other elements are not encoded, since they can be inferred from the DTD. Section 3 describes a tool based on this idea, which was first described by Jansson and Jeuring in the context of data conversion [26, 30]. We use HaXml [51] to translate a DTD to a data type, and we construct generic functions for separating the contents (the strings) and the shape (the constructors) of a value of a data type, and for encoding the shape of a value of a data type using information about the (number of) constructors of the data type.

XML compressors are just one class of XML tools that are easily implemented as generic programs. Other XML tools that can be implemented as generic programs are XML editors, XML databases, and XML version management tools.

Example 3: Zipper. The zipper [24] is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up, or down the tree. For example, the zipper corresponding to the data type Bush, called Loc_Bush, is defined by

```
type Loc_Bush     = (Bush, Context_Bush)
data Context_Bush = Top
                  | BinL Context_Bush Bush
                  | BinR Bush Context_Bush.
```

Using the type of locations we can efficiently navigate through a tree. For example:

$$\begin{aligned}
&\mathit{down\_Bush} &&:: \mathsf{Loc\_Bush} \to \mathsf{Loc\_Bush}\\
&\mathit{down\_Bush}\ (\mathit{Leaf}\ a, c) &&= (\mathit{Leaf}\ a, c)\\
&\mathit{down\_Bush}\ (\mathit{Bin}\ \mathit{tl}\ \mathit{tr}, c) &&= (\mathit{tl}, \mathit{BinL}\ c\ \mathit{tr})\\
&\mathit{right\_Bush} &&:: \mathsf{Loc\_Bush} \to \mathsf{Loc\_Bush}\\
&\mathit{right\_Bush}\ (\mathit{tl}, \mathit{BinL}\ c\ \mathit{tr}) &&= (\mathit{tr}, \mathit{BinR}\ \mathit{tl}\ c)\\
&\mathit{right\_Bush}\ m &&= m.
\end{aligned}$$

The navigation function *down_Bush* moves the focus of attention to the *leftmost* subtree of the current node; *right_Bush* moves the focus to its right sibling.

Huet [24] defines the zipper data structure for rose trees and for the data type Bush, and gives the generic construction in words. In Section 4 we describe the zipper in more detail and show how to define a zipper for an arbitrary data type.

Other Applications of Generic Programming. Besides the applications mentioned in the examples above, there are several application areas in which generic programming can be used.

- *Haskell's* `deriving` *construct.* Haskell's `deriving` construct is used to generate code for for example the equality function, and for functions for reading and showing values of data types. Only the classes `Eq`, `Ord`, `Enum`, `Bounded`, `Show` and `Read` can be derived. The definitions of (equivalents of) the derived functions can be found in the library of Generic Haskell.
- *Compiler functions.* Several functions that are used in compilers are generic functions: garbage collectors, tracers, debuggers, test tools [7, 34], etc.
- *Typed term processing.* Functions like pattern matching, term rewriting and unification are generic functions, and have been implemented as generic functions in [31, 28, 29].

The form and functionality of these applications is exactly determined by the structure of the input data.

Maybe the most common applications of generic programming can be found in functions that traverse data built from rich mutually-recursive data types with many constructors, and which perform computations on a single (or a couple of) constructor(s). For example, consider a function which traverses an abstract syntax tree and returns the free variables in the tree. Only for the variable constructor something special has to be done, in all other cases the variables collected at the children have to be passed on to the parent. This function can be defined as an instance of a Generic Haskell library function *crush* [41], together with a special constructor case for variables [10]. For more examples of generic traversals, see Lämmel and Peyton Jones [35].

The Generic Haskell code for the programs discussed in these lecture notes can be downloaded from the applications page on
`http://www.generic-haskell.org/`.

On Notation. To improve readability, the notation for generic functions we use in these notes slightly differs from the notation used in the first part of these notes [22]. For example, in the first part of these lecture notes we write:

$$\begin{aligned}
&\mathit{equal}\{\!|\mathsf{Char}|\!\} &&= \mathit{eqChar}\\
&\mathit{equal}\{\!|\mathsf{Int}|\!\} &&= \mathit{eqInt}\\
&\mathit{equal}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit}\ \mathit{Unit} &&= \mathit{True}\\
&\mathit{equal}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inl}\ a)\ (\mathit{Inl}\ a') &&= \mathit{eqa}\ a\ a'\\
&\mathit{equal}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inl}\ a)\ (\mathit{Inr}\ b') &&= \mathit{False}\\
&\mathit{equal}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inr}\ b)\ (\mathit{Inl}\ a') &&= \mathit{False}\\
&\mathit{equal}\{\!|{:+:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inr}\ b)\ (\mathit{Inr}\ b') &&= \mathit{eqb}\ b\ b'\\
&\mathit{equal}\{\!|{:*:}|\!\}\ \mathit{eqa}\ \mathit{eqb}\ (a\ {:*:}\ b)\ (a'\ {:*:}\ b') &&= \mathit{eqa}\ a\ a' \wedge \mathit{eqb}\ b\ b'\\
&\mathit{equal}\{\!|\mathsf{Con}\ c|\!\}\ \mathit{eqa}\ (\mathit{Con}\ a)\ (\mathit{Con}\ b) &&= \mathit{eqa}\ a\ b.
\end{aligned}$$

The function *equal* is a generic function which recurses over the type structure of the argument type. The recursion is implicit in the arguments *eqa* and *eqb* in the `:+:` and `:*:` cases. In this part of the lecture notes we will instead write:

$$\begin{aligned}
&\mathit{equal}\{\!|\mathsf{Char}|\!\} &&= \mathit{eqChar}\\
&\mathit{equal}\{\!|\mathsf{Int}|\!\} &&= \mathit{eqInt}\\
&\mathit{equal}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit}\ \mathit{Unit} &&= \mathit{True}\\
&\mathit{equal}\{\!|\mathsf{a}\ {:+:}\ \mathsf{b}|\!\}\ (\mathit{Inl}\ a)\ (\mathit{Inl}\ a') &&= \mathit{equal}\{\!|\mathsf{a}|\!\}\ a\ a'\\
&\mathit{equal}\{\!|\mathsf{a}\ {:+:}\ \mathsf{b}|\!\}\ (\mathit{Inl}\ a)\ (\mathit{Inr}\ b') &&= \mathit{False}\\
&\mathit{equal}\{\!|\mathsf{a}\ {:+:}\ \mathsf{b}|\!\}\ (\mathit{Inr}\ b)\ (\mathit{Inl}\ a') &&= \mathit{False}\\
&\mathit{equal}\{\!|\mathsf{a}\ {:+:}\ \mathsf{b}|\!\}\ (\mathit{Inr}\ b)\ (\mathit{Inr}\ b') &&= \mathit{equal}\{\!|\mathsf{b}|\!\}\ b\ b'\\
&\mathit{equal}\{\!|\mathsf{a}\ {:*:}\ \mathsf{b}|\!\}\ (a\ {:*:}\ b)\ (a'\ {:*:}\ b') &&= \mathit{equal}\{\!|\mathsf{a}|\!\}\ a\ a' \wedge \mathit{equal}\{\!|\mathsf{b}|\!\}\ b\ b'\\
&\mathit{equal}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a)\ (\mathit{Con}\ b) &&= \mathit{equal}\{\!|\mathsf{a}|\!\}\ a\ b.
\end{aligned}$$

Here the recursion over the type structure is explicit: $\mathit{equal}\{\!|\mathsf{a}\ {:*:}\ \mathsf{b}|\!\}$ is expressed in terms of $\mathit{equal}\{\!|\mathsf{a}|\!\}$ and $\mathit{equal}\{\!|\mathsf{b}|\!\}$. We think this style is more readable, especially when a generic function depends on another generic function. Functions written in the latter style can be translated to the former style and vice versa, so no expressiveness is lost or gained; the only difference is readability. The Generic Haskell compiler does not accept explicit recursive functions yet, but probably will do so in the near future. A formal description of Generic Haskell with explicit recursion is given by Löh et al. [37].

Organization. The rest of this paper is organized as follows. Section 2 introduces generic dictionaries, and implements them in Generic Haskell. Section 3 describes XComprez, a compressor for XML documents. Section 4 develops a generic zipper data structure. Section 5 summarizes the main points and concludes.

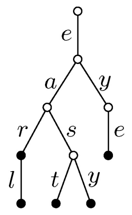
These lecture notes contain exercises. Solutions to the exercises can be found on the webpage for the Generic Haskell project: `www.generic-haskell.org`.

2 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 et al. [11] 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. The material in this section is largely taken from [18].

2.1 Introduction

The concept of a trie was introduced by Thue in 1912 as a means to represent a set of strings, see [33]. 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 { *ear*, *earl*, *east*, *easy*, *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 following data type:

```
data String = Nil | Cons Char String,
```

we can represent string-indexed tries with associated values of type v as follows.

```
data FMap_String v = Null_String
                   | Node_String (Maybe v)
                                 (FMapChar (FMap_String v))
```

Here, *Null_String* represents the empty trie. The first component of the constructor *Node_String* contains the value associated with *Nil*. Its type is Maybe v instead of v since *Nil* 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.

$$
\begin{aligned}
&\textbf{type}\ \mathsf{FMapChar\ v} = [(\mathsf{Char}, \mathsf{v})]\\
&\mathit{lookupChar} && :: \forall \mathsf{v}\,.\,\mathsf{Char} \to \mathsf{FMapChar\ v} \to \mathsf{Maybe\ v}\\
&\mathit{lookupChar}\ c\ [\,] && = \mathit{Nothing}\\
&\mathit{lookupChar}\ c\ ((c', v) : x)\\
&\quad \mid c < c' && = \mathit{Nothing}\\
&\quad \mid c == c' && = \mathit{Just}\ v\\
&\quad \mid c > c' && = \mathit{lookupChar}\ c\ x
\end{aligned}
$$

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, *Cons c s* we look up c in the edge map obtaining a trie, which is then recursively searched for s.

$$
\begin{aligned}
&\mathit{lookup\_String} :: \forall \mathsf{v}\,.\,\mathsf{String} \to \mathsf{FMap\_String\ v} \to \mathsf{Maybe\ v}\\
&\mathit{lookup\_String}\ s\ \mathit{Null\_String} && = \mathit{Nothing}\\
&\mathit{lookup\_String}\ \mathit{Nil}\ (\mathit{Node\_String}\ tn\ tc) && = tn\\
&\mathit{lookup\_String}\ (\mathit{Cons}\ c\ s)\ (\mathit{Node\_String}\ tn\ tc) && =\\
&\quad (\mathit{lookupChar}\ c \diamond \mathit{lookup\_String}\ s)\ tc
\end{aligned}
$$

In the last equation we use monadic composition to take care of the error signal *Nothing*.

Based on work by Wadsworth and others, Connelly et al. [11] 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 *lookup_String* already gives a clue what a suitable generalization might look like: the trie *Node_String tn tc* contains a finite map for each constructor of the data type String; to look up *Cons 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 *Nil*), we extract the associated value.

As a second example, consider the data type of external search trees:

$$\textbf{data}\ \mathsf{Dict} = \mathit{Tip}\ \mathsf{String} \mid \mathit{Node}\ \mathsf{Dict\ String\ Dict}.$$

A trie for external search trees represents a finite map from Dict to some value type v. It is an element of FMap_Dict v given by

$$
\begin{aligned}
\textbf{data}\ \mathsf{FMap\_Dict\ v} = {}& \mathit{Null\_Dict}\\
\mid {}& \mathit{Node\_Dict}\ (\mathsf{FMap\_String\ v})\\
&\qquad (\mathsf{FMap\_Dict}\ (\mathsf{FMap\_String}\ (\mathsf{FMap\_Dict\ v}))).
\end{aligned}
$$

The data type FMap_Dict is a nested data type, since the recursive call on the right hand side, FMap_Dict (FMap_String (FMap_Dict v)), is a substitution instance of the left hand side. Consequently, the look-up function on external search trees requires polymorphic recursion.

$$
\begin{array}{ll}
\mathit{lookup\_Dict} :: \forall \mathsf{v}\,.\,\mathsf{Dict} \to \mathsf{FMap\_Dict\ v} \to \mathsf{Maybe\ v} & \\
\mathit{lookup\_Dict\ d\ Null\_Dict} & = \mathit{Nothing} \\
\mathit{lookup\_Dict}\ (\mathit{Tip\ s})\ (\mathit{Node\_Dict\ tl\ tn}) & = \mathit{lookup\_String\ s\ tl} \\
\mathit{lookup\_Dict}\ (\mathit{Node\ m\ s\ r})\ (\mathit{Node\_Dict\ tl\ tn}) = & \\
\quad (\mathit{lookup\_Dict\ m} \diamond \mathit{lookup\_String\ s} \diamond \mathit{lookup\_Dict\ r})\ \mathit{tn} &
\end{array}
$$

Looking up a node involves two recursive calls. The first, *lookup_Dict m*, is of type $\mathsf{Dict} \to \mathsf{FMap\_Dict\ X} \to \mathsf{Maybe\ X}$ where $\mathsf{X} = \mathsf{FMap\_String\ (FMap\_Dict\ v)}$, which is a substitution instance of the declared type.

Note that it is absolutely necessary that FMap_Dict and *lookup_Dict* are parametric with respect to the codomain of the finite maps. If we restrict the type of *lookup_Dict* to $\mathsf{Dict} \to \mathsf{FMap\_Dict\ T} \to \mathsf{Maybe\ T}$ for some fixed type T, the definition no longer type-checks. This also explains why the construction does not work for the finite set abstraction.

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 renders 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.

$$
\begin{array}{ll}
1 \to_{\mathrm{fin}} \mathsf{v} & \cong \mathsf{v} \\
(\mathsf{t}_1 + \mathsf{t}_2) \to_{\mathrm{fin}} \mathsf{v} & \cong (\mathsf{t}_1 \to_{\mathrm{fin}} \mathsf{v}) \times (\mathsf{t}_2 \to_{\mathrm{fin}} \mathsf{v}) \\
(\mathsf{t}_1 \times \mathsf{t}_2) \to_{\mathrm{fin}} \mathsf{v} & \cong \mathsf{t}_1 \to_{\mathrm{fin}} (\mathsf{t}_2 \to_{\mathrm{fin}} \mathsf{v})
\end{array}
$$

Here, $\mathsf{t} \to_{\mathrm{fin}} \mathsf{v}$ denotes the set of all finite maps from t to v. Note that $\mathsf{t} \to_{\mathrm{fin}} \mathsf{v}$ is sometimes written $\mathsf{v}^{[\mathsf{t}]}$, which explains why these equations are also known as the 'laws of exponentials'.

2.2 Signature

To put the above idea in concrete terms we will define a type-indexed data type FMap, which has the following kind for types t of kind $\star$.

$$\mathsf{FMap}\{\!|\mathsf{t} :: \star|\!\} :: \star \to \star$$

So FMap assigns a type constructor of kind $\star \to \star$ to each key type t of kind $\star$.

We will implement the following operations on tries.

$$
\begin{array}{ll}
\mathit{empty}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
\mathit{isempty}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Bool} \\
\mathit{single}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,(\mathsf{t},\mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \\
\mathit{lookup}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,\mathsf{t} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Maybe\ v} \\
\mathit{insert}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,(\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{t},\mathsf{v}) \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}) \\
\mathit{merge}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,(\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}) \\
\mathit{delete}\{\!|\mathsf{t}|\!\} & :: \forall \mathsf{v}\,.\,\mathsf{t} \to (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v})
\end{array}
$$

The value $empty\{|\mathsf{t}|\}$ is the empty trie. The function $isempty\{|\mathsf{t}|\}$ takes a trie and determines whether or not it is empty. The function $single\{|\mathsf{t}|\}\ (t, v)$ constructs a trie that contains the binding (t, v) as its only element. The function $lookup\{|\mathsf{t}|\}$ takes a key and a trie and looks up the value associated with the key. The function $insert\{|\mathsf{t}|\}$ inserts a new binding into a trie, and $merge\{|\mathsf{t}|\}$ combines two tries. The function $delete\{|\mathsf{t}|\}$ takes a key and a trie, and removes the binding for the key from the trie. The two functions $insert\{|\mathsf{t}|\}$ and $merge\{|\mathsf{t}|\}$ 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 $insert\{|\mathsf{t}|\}$ if the new binding is to override an old binding with the same key. For finite maps of type $\mathsf{FMap}\{|\mathsf{t}|\}\ \mathsf{Int}$ 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 $insert\{|\mathsf{t}|\}$ and $merge\{|\mathsf{t}|\}$ generically for all types!

2.3 Properties

Each of the operations specified in the previous section satisfies a number of laws, which hold generically for all instances of t. These properties formally specify parts of the informal descriptions of the operations given there, and can be proved for the definitions given in the following sections using fixed point induction. See Hinze [19, 21] for examples of proofs of properties of generic functions.

$$\begin{aligned}
lookup\{|\mathsf{t}|\}\ k\ (empty\{|\mathsf{t}|\}) &= Nothing \\
lookup\{|\mathsf{t}|\}\ k\ (single\{|\mathsf{t}|\}\ (k_1, v_1)) &= \mathbf{if}\ k == k_1\ \mathbf{then}\ Just\ v_1\ \mathbf{else}\ Nothing \\
lookup\{|\mathsf{t}|\}\ k\ (merge\{|\mathsf{t}|\}\ c\ t_1\ t_2) &= combine\ c\ (lookup\{|\mathsf{t}|\}\ k\ t_1)\ (lookup\{|\mathsf{t}|\}\ k\ t_2),
\end{aligned}$$

where *combine* combines two values of type Maybe:

$$\begin{aligned}
&combine :: \forall \mathsf{v} . (\mathsf{v} \to \mathsf{v} \to \mathsf{v}) \to (\mathsf{Maybe}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v} \to \mathsf{Maybe}\ \mathsf{v}) \\
&combine\ c\ Nothing\ Nothing \quad = Nothing \\
&combine\ c\ Nothing\ (Just\ v_2)\ = Just\ v_2 \\
&combine\ c\ (Just\ v_1)\ Nothing\ = Just\ v_1 \\
&combine\ c\ (Just\ v_1)\ (Just\ v_2) = Just\ (c\ v_1\ v_2).
\end{aligned}$$

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 $merge\{|\mathsf{t}|\}\ c$ is associative, as well. Furthermore, $empty\{|\mathsf{t}|\}$ is the left and the right unit of $merge\{|\mathsf{t}|\}\ c$:

$$\begin{aligned}
merge\{|\mathsf{t}|\}\ c\ (empty\{|\mathsf{t}|\})\ x &= x \\
merge\{|\mathsf{t}|\}\ c\ x\ (empty\{|\mathsf{t}|\}) &= x \\
merge\{|\mathsf{t}|\}\ c\ x_1\ (merge\{|\mathsf{t}|\}\ c\ x_2\ x_3) &= merge\{|t|\}\ c\ (merge\{|\mathsf{t}|\}\ c\ x_1\ x_2)\ x_3.
\end{aligned}$$

The functions *insert* and *delete* satisfy the following laws: for all k, v, and c,

$$\begin{aligned} &\mathit{insert}\{\!|\mathsf{t}|\!\}\ c\ (k,v)\ (\mathit{empty}\{\!|\mathsf{t}|\!\}) = \mathit{single}\{\!|\mathsf{t}|\!\}\ (k,v) \\ &\mathit{delete}\{\!|\mathsf{t}|\!\}\ k\ (\mathit{single}\{\!|\mathsf{t}|\!\}\ (k,v)) = \mathit{empty}\{\!|\mathsf{t}|\!\}. \end{aligned}$$

The operations satisfy many more laws, but we will omit them here.

2.4 Type-Indexed Tries

We hqe already noted that generalized tries are based on the laws of exponentials.

$$\begin{aligned} 1 \rightarrow_{\text{fin}} \mathsf{v} &\cong \mathsf{v} \\ (\mathsf{t}_1 + \mathsf{t}_2) \rightarrow_{\text{fin}} \mathsf{v} &\cong (\mathsf{t}_1 \rightarrow_{\text{fin}} \mathsf{v}) \times (\mathsf{t}_2 \rightarrow_{\text{fin}} \mathsf{v}) \\ (\mathsf{t}_1 \times \mathsf{t}_2) \rightarrow_{\text{fin}} \mathsf{v} &\cong \mathsf{t}_1 \rightarrow_{\text{fin}} (\mathsf{t}_2 \rightarrow_{\text{fin}} \mathsf{v}) \end{aligned}$$

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* $\bot_{\mathsf{v}}$, see [11]. A finite map is then a function whose value is $\bot_{\mathsf{v}}$ 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, our first example of a type-indexed data type.

$$\begin{aligned} &\mathsf{FMap}\{\!|\mathsf{t} :: \star|\!\} && :: \star \rightarrow \star \\ &\mathsf{FMap}\{\!|\mathsf{Unit}|\!\}\ \mathsf{v} && = \mathsf{Maybe}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{Int}|\!\}\ \mathsf{v} && = \mathit{Patricia}.\mathsf{Dict}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{Char}|\!\}\ \mathsf{v} && = \mathsf{FMapChar}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{t}_1 :\!+\!: \mathsf{t}_2|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ \mathsf{v} \times_{\bullet} \mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v} \\ &\mathsf{FMap}\{\!|\mathsf{t}_1 :\!*\!: \mathsf{t}_2|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ (\mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v}) \\ &\mathsf{FMap}\{\!|\mathsf{Con}\ \mathsf{t}|\!\}\ \mathsf{v} && = \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \end{aligned}$$

Here, $(\times_{\bullet})$ is the type of optional pairs.

$$\textbf{data}\ \mathsf{a} \times_{\bullet} \mathsf{b} = \mathit{Null} \mid \mathit{Pair}\ \mathsf{a}\ \mathsf{b}$$

Instead of optional pairs we could also use ordinary pairs in the definition of FMap:

$$\mathsf{FMap}\{\!|\mathsf{t}_1 :\!+\!: \mathsf{t}_2|\!\}\ \mathsf{v} = \mathsf{FMap}\{\!|\mathsf{t}_1|\!\}\ \mathsf{v} :\!*\!: \mathsf{FMap}\{\!|\mathsf{t}_2|\!\}\ \mathsf{v}.$$

This representation has, however, two major drawbacks: (i) it relies in an essential way on lazy evaluation and (ii) it is inefficient, see [18].

We assume there exists 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* [44]. 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*.

A few remarks are in order. FMap is a type-indexed data type [23]. The only way to construct values of type FMap is by means of the functions in the interface.

Furthermore, in contrast with type-indexed functions, the constructor index Con doesn't mention a constructor description anymore. This is because a type cannot depend on a value, so the constructor description can never be used in the definition of a type-indexed data type.

Since the trie for the unit type is given by $\mathsf{Maybe\ v}$ rather than v itself, tries for isomorphic types are, in general, not isomorphic. We have, for instance, $\mathsf{Unit} \cong \mathsf{Unit} :\!*\!: \mathsf{Unit}$ (ignoring laziness) but $\mathsf{FMap}\{\!|\mathsf{Unit}|\!\}\ \mathsf{v} = \mathsf{Maybe\ v} \not\cong \mathsf{Maybe\ (Maybe\ v)} = \mathsf{FMap}\{\!|\mathsf{Unit} :\!*\!: \mathsf{Unit}|\!\}\ \mathsf{v}$. The trie type $\mathsf{Maybe\ (Maybe\ v)}$ has two different representations of the empty trie: *Nothing* and *Just Nothing*. However, only the first one will be used in our implementation. Similarly, $\mathsf{Maybe\ v} \times_{\bullet} \mathsf{Maybe\ v}$ has two elements, *Null* and *Pair Nothing Nothing*, that represent the empty trie. Again, only the first one will be used.

As mentioned in Section 2.2, the kind of FMap for types of kind $\star$ is $\star \to \star$. For type constructors with higher-order kinds, the kind of FMap looks surprisingly similar to the type of type-indexed functions for higher-order kinds. A trie on the type $\mathsf{List\ a}$ is a trie for the type List, applied to a trie for the type a:

$$\mathsf{FMap}\{\!|\mathsf{f} :: \star \to \star|\!\} :: (\star \to \star) \to (\star \to \star).$$

The 'type' of a type-indexed type is a *kind-indexed kind.* In general, we have:

$$\begin{aligned}
&\mathsf{FMap}\{\!|\mathsf{f} :: \kappa|\!\} && :: \mathsf{FMAP}\{\![\kappa]\!\}\ \mathsf{f} \\
&\mathsf{FMAP}\{\![\kappa :: \Box]\!\} && :: \Box \\
&\mathsf{FMAP}\{\![\star]\!\} && = \star \to \star \\
&\mathsf{FMAP}\{\![\kappa \to \nu]\!\} && = \mathsf{FMAP}\{\![\kappa]\!\} \to \mathsf{FMAP}\{\![\nu]\!\},
\end{aligned}$$

where the box $\Box$ is the type of a kind, a so-called *super-kind.*

Example 1. Let us specialize FMap to the following data types.

$$\begin{aligned}
&\mathbf{data}\ \mathsf{List\ a} && = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{a\ (List\ a)} \\
&\mathbf{data}\ \mathsf{Tree\ a\ b} && = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ \mathsf{(Tree\ a\ b)\ b\ (Tree\ a\ b)} \\
&\mathbf{data}\ \mathsf{Fork\ a} && = \mathit{Fork}\ \mathsf{a\ a} \\
&\mathbf{data}\ \mathsf{Sequ\ a} && = \mathit{EndS} \mid \mathit{ZeroS}\ \mathsf{(Sequ\ (Fork\ a))} \mid \mathit{OneS}\ \mathsf{a\ (Sequ\ (Fork\ a))}
\end{aligned}$$

These types are represented by (see also section 3.1 in the first part of these lecture notes [22]):

$$\begin{aligned}
&\mathsf{List} && = \mathsf{Fix}\ (\Lambda\mathsf{List}\,.\,\Lambda\mathsf{a}\,.\,\mathsf{Unit} :\!+\!: \mathsf{a} :\!*\!: \mathsf{List\ a}) \\
&\mathsf{Tree} && = \mathsf{Fix}\ (\Lambda\mathsf{Tree}\,.\,\Lambda\mathsf{a\ b}\,.\,\mathsf{a} :\!+\!: \mathsf{Tree\ a\ b} :\!*\!: \mathsf{b} :\!*\!: \mathsf{Tree\ a\ b}) \\
&\mathsf{Fork} && = \Lambda\mathsf{a}\,.\,\mathsf{a} :\!*\!: \mathsf{a} \\
&\mathsf{Sequ} && = \mathsf{Fix}\ (\Lambda\mathsf{Sequ}\,.\,\Lambda\mathsf{a}\,.\,\mathsf{Unit} :\!+\!: \mathsf{Sequ\ (Fork\ a)} :\!+\!: \mathsf{a} :\!*\!: \mathsf{Sequ\ (Fork\ a)}).
\end{aligned}$$

Recall that (:*:) binds stronger than (:+:). Consequently, the corresponding trie types are

$$\begin{aligned}
\mathsf{FMap\_List} &= \mathsf{Fix}\ (\Lambda \mathsf{FMap\_List}\,.\,\Lambda \mathsf{fa}\,.\,\mathsf{Maybe} \times_{\bullet} \mathsf{fa} \cdot \mathsf{FMap\_List}\ \mathsf{fa})\\
\mathsf{FMap\_Tree} &= \mathsf{Fix}\ (\Lambda \mathsf{FMap\_Tree}\,.\,\Lambda \mathsf{fa}\ \mathsf{fb}\,.\\
&\qquad \mathsf{fa} \times_{\bullet}\\
&\qquad \mathsf{FMap\_Tree}\ \mathsf{fa}\ \mathsf{fb} \cdot \mathsf{fb} \cdot \mathsf{FMap\_Tree}\ \mathsf{fa}\ \mathsf{fb})\\
\mathsf{FMap\_Fork} &= \Lambda \mathsf{fa}\,.\,\mathsf{fa} \cdot \mathsf{fa}\\
\mathsf{FMap\_Sequ} &= \mathsf{Fix}\ (\Lambda \mathsf{FMap\_Sequ}\,.\,\Lambda \mathsf{fa}\,.\\
&\qquad \mathsf{Maybe} \times_{\bullet}\\
&\qquad \mathsf{FMap\_Sequ}\ (\mathsf{FMap\_Fork}\ \mathsf{fa}) \times_{\bullet}\\
&\qquad \mathsf{fa} \cdot \mathsf{FMap\_Sequ}\ (\mathsf{FMap\_Fork}\ \mathsf{fa})).
\end{aligned}$$

As an aside, note that we interpret $\mathsf{a} \times_{\bullet} \mathsf{b} \times_{\bullet} \mathsf{c}$ as the type of optional triples and not as nested optional pairs:

$$\mathbf{data}\ \mathsf{a} \times_{\bullet} \mathsf{b} \times_{\bullet} \mathsf{c} = \mathit{Null} \mid \mathit{Triple}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c}.$$

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.

```
data FMap_List fa v      = Null_List
                         | Node_List (Maybe v)
                                     (fa (FMap_List fa v))
data FMap_Tree fa fb v = Null_Tree
                         | Node_Tree (fa v)
                                      (FMap_Tree fa fb
                                           (fb (FMap_Tree fa fb v)))
```

These types are the parametric variants of FMap_String and FMap_Dict defined in Section 2.1: we have FMap_String ≈ FMap_List FMapChar (corresponding to String ≈ List Char) and FMap_Dict ≈ FMap_Tree FMap_String FMap_String (corresponding to Dict ≈ Tree String String). Things become interesting if we consider nested data types.

```
data FMap_Fork fa v = Node_Fork (fa (fa v))
data FMap_Sequ fa v = Null_Sequ
                    | Node_Sequ (Maybe v)
                                (FMap_Sequ (FMap_Fork fa) v)
                                (fa (FMap_Sequ (FMap_Fork fa) v))
```

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 FMap_Sequ is a second-order nest since the parameter fa of kind $\star \to \star$ is changed in the recursive calls. By contrast, FMap_Tree 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—$\mathit{OneS}\ \mathsf{a}\ (\mathsf{Sequ}\ (\mathsf{Fork}\ \mathsf{a}))$ becomes $\mathit{OneS}\ (\mathsf{Sequ}\ (\mathsf{Fork}\ \mathsf{a}))\ \mathsf{a}$—then the third component of $\mathsf{FMap\_Sequ}$ has type $\mathsf{FMap\_Sequ}\ (\mathsf{FMap\_Fork}\ \mathsf{fa})\ (\mathsf{fa}\ \mathsf{v})$ and since both fa and v are instantiated, $\mathsf{FMap\_Sequ}$ is consequently both a first- and a second-order nest.

2.5 Empty Tries

The empty trie is defined as follows.

$$\begin{array}{ll}
\textbf{type}\ \mathsf{Empty}\{[\star]\}\ \mathsf{t} & = \forall \mathsf{v}\,.\,\mathsf{FMap}\{|\mathsf{t}|\}\ \mathsf{v} \\
\textbf{type}\ \mathsf{Empty}\{[\kappa \to \nu]\}\ \mathsf{t} & = \forall \mathsf{a}\,.\,\mathsf{Empty}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Empty}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})
\end{array}$$

$$\begin{array}{lll}
\mathit{empty}\{|\mathsf{t} :: \kappa|\} & :: & \mathsf{Empty}\{[\kappa]\}\ \mathsf{t} \\
\mathit{empty}\{|\mathsf{Unit}|\} & = & \mathit{Nothing} \\
\mathit{empty}\{|\mathsf{Char}|\} & = & [\,] \\
\mathit{empty}\{|\mathsf{Int}|\} & = & \mathit{Patricia.empty} \\
\mathit{empty}\{|\mathsf{a} :+: \mathsf{b}|\} & = & \mathit{Null} \\
\mathit{empty}\{|\mathsf{a} :*: \mathsf{b}|\} & = & \mathit{empty}\{|\mathsf{a}|\} \\
\mathit{empty}\{|\mathsf{Con}\ c\ \mathsf{a}|\} & = & \mathit{empty}\{|\mathsf{a}|\}
\end{array}$$

The definition already illustrates several interesting aspects of programming with generalized tries. First, the explicit polymorphic type of *empty* is necessary to make the definition work. Consider the line $\mathit{empty}\{|\mathsf{a} :*: \mathsf{b}|\}$, which is of type $\forall \mathsf{v}\,.\,\mathsf{FMap}\{|\mathsf{a}|\}\ (\mathsf{FMap}\{|\mathsf{b}|\}\ \mathsf{v})$. It is defined in terms of $\mathit{empty}\{|\mathsf{a}|\}$, which is of type $\forall \mathsf{v}\,.\,\mathsf{FMap}\{|\mathsf{a}|\}\ \mathsf{v}$. That means that $\mathit{empty}\{|\mathsf{a}|\}$ is used polymorphically. In other words, *empty* makes use of polymorphic recursion!

Suppose we want to define a function *emptyI* that is almost the same as the function *empty*, but uses a different value, say *emptyIntTrie*, for the empty trie for integers. The definition of FMap says that *emptyIntTrie* has to be a *Patricia* tree, but that might be changed in the definition of FMap. Then we can use a default case [10] to define *emptyI* in terms of *empty* as follows:

$$\begin{array}{ll}
\mathit{emptyI}\{|\mathsf{t} :: \kappa|\} & :: \mathit{Empty}\ (\kappa)\ \mathsf{t} \\
\mathit{emptyI}\{|\mathsf{Int}|\} & = \mathit{emptyIntTrie} \\
\mathit{emptyI}\{|\mathsf{a}|\} & = \mathit{empty}\{|\mathsf{a}|\}.
\end{array}$$

So the function *emptyI* is equal to the function *empty* in all cases except for the Int case, where it uses a special empty trie.

Example 2. Let us specialize *empty* to lists and binary random-access lists.

$$\begin{array}{l}
\mathit{empty\_List} :: \forall \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v}\,.\,\mathsf{FMap\_List}\ \mathsf{fa}\ \mathsf{v}) \\
\mathit{empty\_List}\ \mathit{ea} = \mathit{Null\_List} \\
\mathit{empty\_Fork} :: \forall \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v}\,.\,\mathsf{FMap\_Fork}\ \mathsf{fa}\ \mathsf{v}) \\
\mathit{empty\_Fork}\ \mathit{ea} = \mathit{Node\_Fork}\ \mathit{ea} \\
\mathit{empty\_Sequ} :: \forall \mathsf{fa}\,.\,(\forall \mathsf{w}\,.\,\mathsf{fa}\ \mathsf{w}) \to (\forall \mathsf{v}\,.\,\mathsf{FMap\_Sequ}\ \mathsf{fa}\ \mathsf{v}) \\
\mathit{empty\_Sequ}\ \mathit{ea} = \mathit{Null\_Sequ}
\end{array}$$

The second function, *empty_Fork*, illustrates the polymorphic use of the parameter: *ea* has type ∀w . fa w but is used as an element of fa (fa w). The functions *empty_List* and *empty_Sequ* show that the 'mechanically' generated definitions can sometimes be slightly improved: the argument *ea* is not needed.

The function *isempty*{|*t*|} takes a trie and determines whether it is empty.

```
type IsEmpty{[⋆]} t      = ∀v . FMap{|t|} v → Bool
type IsEmpty{[κ → ν]} t = ∀a . IsEmpty{[κ]} a → IsEmpty{[ν]} (t a)
```

```
isempty{|t :: κ|}          :: IsEmpty{[κ]} t
isempty{|Unit|} v          = isNothing v
isempty{|Char|} m          = null m
isempty{|Int|} m           = Patricia.isempty m
isempty{|a :+: b|} Null    = True
isempty{|a :+: b|} d       = False
isempty{|a :*: b|} d       = isempty{|a|} d
isempty{|Con c a|} d       = isempty{|a|} d
```

Function *isempty* assumes that tries are in 'normal form', so the empty trie is always represented by *Null*, and not, for example, by *Pair Null Null*.

Example 3. Let us specialize *isempty* to lists and binary random-access lists.

```
isempty_List                             :: ∀fa . (∀w . fa w → Bool) →
                                                  (∀v . FMap_List fa v → Bool)
isempty_List iea Null_List               = True
isempty_List iea (Node_List tn tc)       = False
isempty_Fork                             :: ∀fa . (∀w . fa w → Bool) →
                                                  (∀v . FMap_Fork fa v → Bool)
isempty_Fork iea (Node_Fork tf)          = iea tf
isempty_Sequ                             :: ∀fa . (∀w . fa w → Bool) →
                                                  (∀v . FMap_Sequ fa v → Bool)
isempty_Sequ iea Null_Sequ               = True
isempty_Sequ iea (Node_Sequ tv tf ts)    = False
```

2.6 Singleton Tries

The function *single*{|t|} (t, v) constructs a trie that contains the binding (t, v) as its only element. To construct a trie in the sum case, we have to return a *Pair*, of which only one component is inhabited. The other component is the empty trie. This means that *single depends on empty*. Generic Haskell supports dependencies, so we can use both the empty trie and the single trie in the sum, product, and constructor cases of function *single*. The dependency shows in the type of the function *single*: on higher-order kinds, the type mentions the type of *empty*.

```
type Single{[⋆]} t      = ∀v . (t, v) → FMap{|t|} v
type Single{[κ → ν]} t = ∀a . Empty{[κ]} a → Single{[κ]} a → Single{[ν]} (t a)
```

Plain generic functions can be seen as catamorphisms [38, 42] over the structure of data types. With dependencies, we also get the power of paramorphisms [40].

$$\begin{array}{ll}
\mathit{single}\{\!|\mathsf{t} :: \kappa|\!\} & :: \mathsf{Single}\{[\kappa]\}\ \mathsf{t} \\
\mathit{single}\{\!|\mathsf{Unit}|\!\}\ (\mathit{Unit}, v) & = \mathit{Just}\ v \\
\mathit{single}\{\!|\mathsf{Char}|\!\}\ (c, v) & = [(c, v)] \\
\mathit{single}\{\!|\mathsf{Int}|\!\}\ (i, v) & = \mathit{Patricia.single}\ (i, v) \\
\mathit{single}\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (\mathit{Inl}\ a, v) & = \mathit{Pair}\ (\mathit{single}\{\!|\mathsf{a}|\!\}\ (a, v))\ (\mathit{empty}\{\!|b|\!\}) \\
\mathit{single}\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (\mathit{Inr}\ b, v) & = \mathit{Pair}\ (\mathit{empty}\{\!|\mathsf{a}|\!\})\ (\mathit{single}\{\!|\mathsf{b}|\!\}\ (b, v)) \\
\mathit{single}\{\!|\mathsf{a} :\!*\!: \mathsf{b}|\!\}\ (a :\!*\!: b, v) & = \mathit{single}\{\!|\mathsf{a}|\!\}\ (a, \mathit{single}\{\!|\mathsf{b}|\!\}\ (b, v)) \\
\mathit{single}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ b, v) & = \mathit{single}\{\!|\mathsf{a}|\!\}\ (b, v)
\end{array}$$

Example 4. Let us again specialize the generic function to lists and binary random-access lists.

```
single_List :: ∀k fa . (∀w . fa w) → (∀w . (k, w) → fa w)
                         → (∀v . (List k, v) → FMap_List fa v)
single_List ea sa (Nil, v)          = Node_List (Just v) ea
single_List ea sa (Cons k ks, v)    =
  Node_List Nothing (sa (k, single_List ea sa (ks, v)))
single_Fork :: ∀k fa . (∀w . fa w) → (∀w . (k, w) → fa w)
                         → (∀v . (Fork k, v) → FMap_Fork fa v)
single_Fork ea sa (Fork k1 k2, v) = Node_Fork (sa (k1, sa (k2, v)))
single_Sequ :: ∀k fa . (∀w . fa w) → (∀w . (k, w) → fa w)
                         → (∀v . (Sequ k, v) → FMap_Sequ fa v)
single_Sequ ea sa (EndS, v)         = Node_Sequ (Just v) Null_Sequ ea
single_Sequ ea sa (ZeroS s, v)      =
    Node_Sequ Nothing
              (single_Sequ (empty_Fork ea) (single_Fork ea sa) (s, v))
              ea
single_Sequ ea sa (OneS k s, v)     =
    Node_Sequ Nothing
              Null_Sequ
              (sa (k, single_Sequ (empty_Fork ea) (single_Fork ea sa) (s, v)))
```

Again, we can simplify the 'mechanically' generated definitions: since the definition of Fork does not involve sums, *single_Fork* does not require its first argument, *ea*, which can be safely removed.

2.7 Look up

The look-up function implements the scheme discussed in Section 2.1.

```
type Lookup{[⋆]} t       = ∀v . t → FMap{|t|} v → Maybe v
type Lookup{[κ → ν]} t = ∀a . Lookup{[κ]} a → Lookup{[ν]} (t a)
```

```
lookup{|t :: κ|}                                 :: Lookup{[κ]} t
lookup{|Unit|} Unit fm                           = fm
lookup{|Char|} c fm                              = lookupChar c fm
lookup{|Int|} i fm                               = Patricia.lookup i fm
lookup{|a :+: b|} t Null                         = Nothing
lookup{|a :+: b|} (Inl a) (Pair fma fmb)         = lookup{|a|} a fma
lookup{|a :+: b|} (Inr b) (Pair fma fmb)         = lookup{|b|} b fmb
lookup{|a :*: b|} (a :*: b) fma                  = (lookup{|a|} a ◇ lookup{|b|} b) fma
lookup{|Con d a|} (Con b) fm                     = lookup{|a|} b fm
```

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 monadic composition.

Example 5. Specializing *lookup*{|t|} to concrete instances of t is by now probably a matter of routine.

```
lookup_List :: ∀k fa . (∀w . k → fa w → Maybe w)
                     → (∀v . List k → FMap_List fa v → Maybe v)
lookup_List la ks Null_List                    = Nothing
lookup_List la Nil (Node_List tn tc)           = tn
lookup_List la (Cons k ks) (Node_List tn tc)   = (la k ◇ lookup_List la ks) tc
lookup_Fork :: ∀k fa . (∀w . k → fa w → Maybe w)
                     → (∀v . Fork k → FMap_Fork fa v → Maybe v)
lookup_Fork la (Fork k₁ k₂) (Node_Fork tf)     = (la k₁ ◇ la k₂) tf
lookup_Sequ :: ∀fa k . (∀w . k → fa w → Maybe w)
                     → (∀v . Sequ k → FMap_Sequ fa v → Maybe v)
lookup_Sequ la s Null_Sequ                     = Nothing
lookup_Sequ la EndS (Node_Sequ te tz to)       = te
lookup_Sequ la (ZeroS s) (Node_Sequ te tz to)  =
   lookup_Sequ (lookup_Fork la) s tz
lookup_Sequ la (OneS a s) (Node_Sequ te tz to) =
   (la a ◇ lookup_Sequ (lookup_Fork la) s) to
```

The function *lookup_List* generalizes *lookup_String* defined in Section 2.1; we have *lookup_String* $\approx$ *lookup_List lookupChar*.

2.8 Inserting and Merging

Insertion is defined in terms of *merge* and *single*.

```
insert{|t :: ⋆|}        :: ∀v . (v → v → v) → (t, v) → FMap{|t|} v → FMap{|t|} v
insert{|t|} c (x, v) d = merge{|t|} c (single{|t|} (x, v)) d
```

Function *insert* is defined as a *generic abstraction*. A generic abstraction is a generic function that is defined in terms of another generic function. The abstracted type parameter is, however, restricted to types of a fixed kind. In the above case, *insert* only works for types of kind ⋆. In the exercise at the end of this section you will define *insert* as a generic function that works for type constructors of all kinds.

Merging two tries is surprisingly simple. Given the function *combine* defined in section 2.3, and a function for merging two association lists

```
mergeChar          :: ∀v . (v → v → v) →
                          (FMapChar v → FMapChar v → FMapChar v)
mergeChar c [ ] x′ = x′
mergeChar c x [ ]  = x
mergeChar c ((k, v) : x) ((k′, v′) : x′)
      | k < k′     = (k, v) : mergeChar c x ((k′, v′) : x′)
      | k == k′    = (k, c v v′) : mergeChar c x x′
      | k > k′     = (k′, v′) : mergeChar c ((k, v) : x) x′,
```

we can define *merge* as follows.

```
type Merge{[⋆]} t = ∀v .
  (v → v → v) → FMap{|t|} v → FMap{|t|} v → FMap{|t|} v
type Merge{[κ → ν]} t = ∀a . Merge{[κ]} a → Merge{[ν]} (t a)
```

```
merge{|t :: κ|}               :: Merge{[κ]} t
merge{|Unit|} c v v′          = combine c v v′
merge{|Char|} c fm fm′        = mergeChar c fm′ fm
merge{|Int|} c fm fm′         = Patricia.merge c fm′ fm
merge{|Con d a|} c e e′       = merge{|a|} c e e′
```

For the sum case, we have to distinguish between empty and nonempty tries:

```
merge{|a :+: b|} c d Null                  = d
merge{|a :+: b|} c Null d                  = d
merge{|a :+: b|} c (Pair x y) (Pair v w) =
    Pair (merge{|a|} c x v) (merge{|b|} c y w).
```

The most interesting equation is the product case. The tries d and d' are of type FMap{|a|} (FMap{|b|} v). To merge them we can recursively call *merge*{|a|}; we must, however, supply a combining function of type ∀v . FMap{|b|} v → FMap{|b|} v → FMap{|b|} v. A moment's reflection reveals that *merge*{|b|} c is the desired combining function.

```
merge{|a :*: b|} c d d′ = merge{|a|} (merge{|b|} c) d d′
```

The definition of *merge* shows that it is sometimes necessary to implement operations more general than immediately needed. If Merge{[⋆]} t had been the simpler type ∀v . FMap{|t|} v → FMap{|t|} v → FMap{|t|} v, then we would not have been able to give a defining equation for :*:.

Example 6. 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).

```
merge_List :: ∀fa . (∀w . (w → w → w) → (fa w → fa w → fa w))
                  → (∀v . (v → v → v)
                     → FMap_List fa v → FMap_List fa v → FMap_List fa v)
merge_List ma c Null_List t                                 = t
merge_List ma c t Null_List                                 = t
merge_List ma c (Node_List tn tc) (Node_List tn' tc')       =
  Node_List (combine c tn tn')
            (ma (merge_List ma c) tc tc')
merge_Fork :: ∀fa . (∀w . (w → w → w) → (fa w → fa w → fa w))
                  → (∀v . (v → v → v)
                     → FMap_Fork fa v → FMap_Fork fa v → FMap_Fork fa v)
merge_Fork ma c (Node_Fork tf) (Node_Fork tf')              =
  Node_Fork (ma (ma c) tf tf')
merge_Sequ :: ∀fa . (∀w . (w → w → w) → (fa w → fa w → fa w))
                  → (∀v . (v → v → v)
                     → FMap_Sequ fa v → FMap_Sequ fa v → FMap_Sequ fa v)
merge_Sequ ma c Null_Sequ t                                 = t
merge_Sequ ma c t Null_Sequ                                 = t
merge_Sequ ma c (Node_Sequ te tz to) (Node_Sequ te' tz' to') =
  Node_Sequ (combine c te te')
            (merge_Sequ (merge_Fork ma) c tz tz')
            (ma (merge_Sequ (merge_Fork ma) c) to to')
```

2.9 Deleting

Function *delete*{|t|} takes a key and a trie, and removes the binding for the key from the trie. For the Char case we need a help function that removes an element from an association list:

```
deleteChar :: ∀v . Char → FMapChar v → FMapChar v,
```

and similarly for the Int case. Function *delete* is defined as follows:

```
delete{|t :: κ|}           :: Delete{[κ]} t
delete{|Unit|} Unit fm     = Nothing
delete{|Char|} c fm        = deleteChar c fm
delete{|Int|} i fm         = Patricia.delete i fm.
```

All cases except the product case are straightforward. In the product case, we have to remove a binding for a product $(a :*: b)$. We do this by using a to lookup the trie d in which there is a binding for b. Then we remove the binding for b in d, obtaining a trie d'. If d' is empty, then we delete the complete binding for a in d, otherwise we insert the binding (a, d') in the original trie d. Here we pass as a combining function $\lambda x\ y \to x$, which overwrites existing bindings in a trie. From this description it follows that the function *delete* depends on the functions *lookup*, *insert* (which depends on function *empty*), and *isempty*.

Here we need the kind-indexed typed version of function *insert*, as defined in the exercise at the end of this section.

```
type Delete{[⋆]} t      = ∀v . t → FMap{|t|} v → FMap{|t|} v
type Delete{[κ → ν]} t =  ∀a . Lookup{[κ]} a
                              → Insert{[κ]} a
                              → IsEmpty{[κ]} a
                              → Empty{[κ]} a
                              → Delete{[κ]} a
                              → Delete{[ν]} (t a)
```

```
delete{|a :+: b|} t Null               = Null
delete{|a :+: b|} (Inl a) (Pair x y)  = Pair (delete{|a|} a x) y
delete{|a :+: b|} (Inr b) (Pair x y)  = Pair x (delete{|b|} b y)
delete{|a :*: b|} (a :*: b) d         =
  case (lookup{|a|} a ◇ delete{|b|} b) d of
    Nothing                            → d
    Just d′ | isempty{|b|} d′          → delete{|a|} a d
            | otherwise                → insert{|a|} (λx y → x) (a, d′) d
delete{|Con c a|} (Con b) d           = delete{|a|} b d
```

Function *delete* should also maintain the invariant that the empty trie is represented by *Null*, and not by *Pair Null Null*, for example. It is easy to adapt the above definition such that this invariant is maintained.

Since the type of *delete* is rather complex because of the dependencies, we only give the instance of *delete* on List.

```
delete_List :: ∀k fa . (∀w . k → fa w → Maybe w)
                     → (∀w . (w → w → w) → (k, w) → fa w → fa w)
                     → (∀w . fa w → Bool)
                     → (∀w . fa w)
                     → (∀w . (k → fa w → fa w))
                     → (∀v . List k → FMap_List fa v → FMap_List fa v)
delete_List la ia iea ea da Nil Null_List                 = Null_List
delete_List la ia iea ea da Nil (Node_List tn tc)         = Node_List Nothing tc
delete_List la ia iea ea da (Cons a as) Null_List         = Null_List
delete_List la ia iea ea da (Cons a as) (Node_List tn tc) =
  case (la a ◇ delete_List la ia iea ea da as) tc of
    Nothing → tc
    Just tc′ | iea tc′ → da a tc
             | otherwise → ia (λx y → x) (a, tc′) tc
```

2.10 Related Work

Knuth [33] attributes the idea of a trie to Thue who introduced it in a paper about strings that do not contain adjacent repeated substrings [47]. De la

Briandais [13] recommended tries for computer searching. The generalization of tries from strings to elements built according to an arbitrary signature was discovered by Wadsworth [50] and others independently since. Connelly et al. [11] 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 [43]. 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.

Exercise 1. Define function *insert* as a generic function with a kind-indexed kind. You can download the code for the functions described in this section from `http://www.generic-haskell.org`, including the solution to this exercise. You might want to avoid looking at the implementation of *insert* while solving this exercise.

Exercise 2. Define a function *update*, which updates a binding in a trie.

$$\mathit{update}\{\!|\mathsf{t}|\!\} :: \forall \mathsf{v}\,.\,(\mathsf{t},\mathsf{v}) \to \mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v} \to \mathsf{Maybe}\ (\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v})$$

If there is no binding for the value of type t in trie of type $\mathsf{FMap}\{\!|\mathsf{t}|\!\}\ \mathsf{v}$, *update* returns *Nothing*.

3 XComprez: A Generic XML Compressor

The extensible markup language XML is a popular standard for describing documents with markup (or structure). XML documents may become (very) large because of the markup that is added to the content. A lot of diskspace and bandwidth is used to store and send XML documents. XML compressors reduce the size of an XML document, sometimes by a considerable factor. This section describes a generic XML compressor based on the ideas described in the context of data conversion by Jansson and Jeuring [26, 30].

This section shows how an XML compressor is implemented as a generic program, and it briefly discusses which other classes of XML tools would profit from an implementation as a generic program. The example shows how Generic Haskell can be used to implement XML tools whose behaviour depends on the DTD or Schema of the input XML document. Example tools include XML editors, databases, and compressors.

Generic Haskell is ideally suited for implementing XML tools:

– Knowledge of the DTD can be used to provide more precise functionality, such as manipulations of an XML document that preserve validity in an XML editor, or better compression in an XML compressor.
– Generic Haskell programs are typed. Consequently, valid documents are transformed to valid documents, possibly structured according to another DTD. Thus Generic Haskell supports constructing type correct XML tools.

- The generic features of Generic Haskell make XML tools easier to implement in a surprisingly small amount of code.
- The Generic Haskell compiler may perform all kinds of advanced optimisations on the code, such as partial evaluation or deforestation, which are difficult to conceive or implement by an XML tool developer.

3.1 Implementing an XML Compressor as a Generic Program

We have implemented an XML compressor, called XComprez, as a generic program. XComprez separates structure from contents, compresses the structure using knowledge about the DTD, and compresses the contents using the Unix compress utility [52]. Thus we replace each element, or rather, the pair of open and close keywords of the element, by the minimal number of bits required for the element given the DTD. We distinguish four components in the tool:

- a component that translates a DTD to a data type,
- a component that separates a value of a data type into its structure and its contents,
- a component that encodes the structure replacing constructors by bits,
- and a component for compressing the contents.

Of course, we have also implemented a decompressor, but since it is very similar to the compressor, we omit its description. See the website for XComprez [32] for the latest developments on XComprez. The Generic Haskell source code for XComprez can be obtained from the website.

Translating a DTD to a Data Type. A DTD can be translated to one or more Haskell data types. For example, the following DTD:

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST book lang (English | Dutch) #REQUIRED> ,
```

can be translated to the following data types:

```
data Book          = Book Book_Attrs Title Author Date [Chapter]
data Book_Attrs    = Book_Attrs{ bookLang :: Lang}
data Lang          = English | Dutch
newtype Title      = Title String
newtype Author     = Author String
newtype Date       = Date String
newtype Chapter    = Chapter String.
```

We have used the Haskell library HaXml [51], in particular the functionality in the module DtdToHaskell to obtain a data type from a DTD, together with functions for reading (parsing) and writing (pretty printing) valid XML documents to and from a value of the generated data type. For example, the following value of the above DTD:

```
<book lang="English">
<title>  Dead Famous  </title>
<author> Ben Elton    </author>
<date>   2001         </date>
<chapter>Introduction </chapter>
<chapter>Preliminaries</chapter>
</book> ,
```

is translated to the following value of the data type Book:

```
Book Book_Attrs { bookLang = English}
     (Title "␣␣Dead␣Famous␣␣")
     (Author "␣Ben␣Elton␣␣␣␣")
     (Date "␣␣␣2001␣␣␣␣␣␣␣␣␣")
     [Chapter "Introduction␣"
     , Chapter "Preliminaries"
     ].
```

An element is translated to a value of a data type using just constructors and no labelled fields. An attribute is translated to a value that contains a labelled field for the attribute. Thus we can use the Generic Haskell constructs Con and Label to distinguish between elements and attributes in generic programs. We have not introduced the Label construct in these lecture notes. It it used to represent record labels in data types, and is very similar to the Con construct.

Separating Structure and Contents. The contents of an XML document is obtained by extracting all `PCData` (Parsable Character Data: characters without tags) and all `CData` (Character Data: characters with possibly tags, starting with '`<![CDATA[`' ending in '`]]>`') from the document. In Generic Haskell, the contents of a value of a data type is obtained by extracting all strings from the value. For the above example value, we obtain the following result:

```
["␣␣Dead␣Famous␣␣"
,"␣Ben␣Elton␣␣␣␣"
,"␣␣␣2001␣␣␣␣␣␣␣␣␣"
,"Introduction␣"
,"Preliminaries"
].
```

The generic function *extract*, which extracts all strings from a value of a data type, is defined as follows:

```
type Extract{[⋆]} t          = t → [String]
type Extract{[κ → ν]} t      = ∀a . Extract{[κ]} a → Extract{[ν]} (t a)
extract{|t :: κ|}            :: Extract{[κ]} t
extract{|Unit|} Unit         = [ ]
extract{|String|} s          = [s]
extract{|a :+: b|} (Inl x)   = extract{|a|} x
extract{|a :+: b|} (Inr y)   = extract{|b|} y
extract{|a :*: b|} (x :*: y) = extract{|a|} x ++ extract{|b|} y
extract{|Con c a|} (Con b)   = extract{|a|} b.
```

Note that it is possible to give special instances of a generic function on a particular type, as with *extract*{|String|} in the above definition. Furthermore, because DtdToHaskell translates any DTD to a data type of kind ⋆, we could have defined *extract* just on data types of kind ⋆. However, higher-order kinds pose no problems. Finally, the operator ++ in the product case is a source of inefficiency. It can be removed using a standard transformation, see Exercise 8 in the first part of these lecture notes.

The structure from an XML document is obtained by removing all `PCData` and `CData` from the document. In Generic Haskell, the structure, or *shape*, of a value is obtained by replacing all strings by empty tuples. Thus we obtain a value that has a different type, in which occurrences of the type String have been replaced by the type (). This is another example of a type-indexed data type [23]. For example, the type we obtain from the data type Book is isomorphic to the following data type:

```
data ShapeBook           = ShapeBook ShapeBook_Attrs
                                     ShapeTitle
                                     ShapeAuthor
                                     ShapeDate
                                     [ShapeChapter]
data ShapeBook_Attrs     = ShapeBook_Attrs{ bookLang :: ShapeLang }
data ShapeLang           = SHAPEEnglish | SHAPEDutch
newtype ShapeTitle       = ShapeTitle ()
newtype ShapeAuthor      = ShapeAuthor ()
newtype ShapeDate        = ShapeDate ()
newtype ShapeChapter     = ShapeChapter (),
```

and the structure of the example value is

```
shapeBook = ShapeBook  ( ShapeBook_Attrs{ bookLang = SHAPEEnglish })
                       ( ShapeTitle ())
                       ( ShapeAuthor ())
                       ( ShapeDate ())
                       [ ShapeChapter ()
                       , ShapeChapter ()
                       ]  .
```

The type-indexed data type Shape replaces occurrences of String in a data type by Unit.

$$\begin{aligned}
\mathsf{Shape}\{\!|\mathsf{Unit}|\!\} &= \mathsf{Unit}\\
\mathsf{Shape}\{\!|\mathsf{String}|\!\} &= ()\\
\mathsf{Shape}\{\!|\mathsf{a} :+: \mathsf{b}|\!\} &= \mathsf{Shape}\{\!|\mathsf{a}|\!\} :+: \mathsf{Shape}\{\!|\mathsf{b}|\!\}\\
\mathsf{Shape}\{\!|\mathsf{a} :*: \mathsf{b}|\!\} &= \mathsf{Shape}\{\!|\mathsf{a}|\!\} :*: \mathsf{Shape}\{\!|\mathsf{b}|\!\}\\
\mathsf{Shape}\{\!|\mathsf{Con\ a}|\!\} &= \mathsf{Con}\ (\mathsf{Shape}\{\!|\mathsf{a}|\!\})
\end{aligned}$$

The generic function *shape* returns the shape of a value of any data type. It has the following kind-indexed type.

$$\begin{aligned}
&\textbf{type}\ \mathsf{Shape}\{[\star]\}\ \mathsf{t} &&= \mathsf{t} \to \mathsf{Shape}\{\!|\mathsf{t}|\!\}\\
&\textbf{type}\ \mathsf{Shape}\{[\kappa \to \nu]\}\ \mathsf{t} &&= \forall \mathsf{a} \,.\, \mathsf{Shape}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Shape}\{[\nu]\}\ (\mathsf{t\ a})
\end{aligned}$$

Note that we use the same name both for the kind-indexed type of function *shape*, as well as the type-indexed data type Shape. They can be distinguished based on their index.

$$\begin{aligned}
&\mathit{shape}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Shape}\{[\kappa]\}\ \mathsf{t}\\
&\mathit{shape}\{\!|\mathsf{Unit}|\!\}\ \mathit{Unit} &&= \mathit{Unit}\\
&\mathit{shape}\{\!|\mathsf{String}|\!\}\ s &&= ()\\
&\mathit{shape}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inl}\ a) &&= \mathit{Inl}\ (\mathit{shape}\{\!|\mathsf{a}|\!\}\ a)\\
&\mathit{shape}\{\!|\mathsf{a} :+: \mathsf{b}|\!\}\ (\mathit{Inr}\ b) &&= \mathit{Inr}\ (\mathit{shape}\{\!|\mathsf{b}|\!\}\ b)\\
&\mathit{shape}\{\!|\mathsf{a} :*: \mathsf{b}|\!\}\ (a :*: b) &&= (\mathit{shape}\{\!|\mathsf{a}|\!\}\ a :*: \mathit{shape}\{\!|\mathsf{b}|\!\}\ b)\\
&\mathit{shape}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ b) &&= \mathit{Con}\ (\mathit{shape}\{\!|\mathsf{a}|\!\}\ b)
\end{aligned}$$

Given the shape and the contents (obtained by means of function *extract*) of a value we obtain the original value by means of function *insert*:

$$\mathit{insert}\{\!|\mathsf{t} :: \star|\!\} :: \mathsf{Shape}\{\!|\mathsf{t}|\!\} \to [\mathsf{String}] \to \mathsf{t}.$$

The generic definition (with a kind-indexed type) of *insert* is left as an exercise.

Encoding Constructors. The constructor of a value is encoded as follows. First calculate the number n of constructors of the data type. Then calculate the position of the constructor in the list of constructors of the data type. Finally, replace the constructor by the bit representation of its position, using $\log_2\ n$ bits. For example, in a data type with 6 constructors, the third constructor is encoded by `010`. We start counting with 0. Furthermore, a value of a data type with a single constructor is represented using 0 bits. Consequently, the values of all types except for String and Lang in the running example are represented using 0 bits.

We assume there exists a function *constructorPosition* which given a constructor returns a pair of integers: its position in the list of constructors of the data type, and the number of constructors of the data type.

$$\mathit{constructorPosition} :: \mathsf{ConDescr} \to (\mathsf{Int}, \mathsf{Int})$$

Function *constructorPosition* can be defined by means of function *constructors*, which returns the constructor descriptions of a data type. This function is defined in the module *Collect*, which can be found in the library of Generic Haskell.

$$\mathit{constructors}\{\!|\mathsf{t} :: \star|\!\} :: [\mathsf{ConDescr}]$$

Function *constructors* is defined for arbitrary kinds in module Collect. We omit the definitions of both function *constructors* and function *constructorPosition*.

The function *encode* takes a value, and encodes it as a value of type Bin, a list of bits, defined in the first part of these lecture notes. The difference with the function *encode* that is defined in the first part of these lecture notes is that here we encode the constructors of the value, and not the choices made in the sum. On average, the function *encode* given here compresses much better than the function *encode* from the first part of these lecture notes.

$$\begin{aligned}
&\mathbf{type}\ \mathsf{Encode}\{[\star]\}\ \mathsf{t} &&= \mathsf{Shape}\{\!|\mathsf{t}|\!\} \to \mathsf{Bin}\\
&\mathbf{type}\ \mathsf{Encode}\{[\kappa \to \nu]\}\ \mathsf{t} &&= \forall \mathsf{a}\,.\, \mathsf{Encode}\{[\kappa]\}\ \mathsf{a} \to \mathsf{Encode}\{[\nu]\}\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

The interesting case in the definition of function *encode* is the constructor case. We first give the simple cases:

$$\begin{aligned}
&\mathit{encode}\{\!|\mathsf{t} :: \kappa|\!\} &&:: \mathsf{Encode}\{[\kappa]\}\ \mathsf{t}\\
&\mathit{encode}\{\!|\mathsf{Unit}|\!\}\ \_ &&= [\,]\\
&\mathit{encode}\{\!|\mathsf{String}|\!\}\ \_ &&= [\,]\\
&\mathit{encode}\{\!|\mathsf{a} :\!*\!: \mathsf{b}|\!\}\ (a :\!*\!: b) &&= \mathit{encode}\{\!|\mathsf{a}|\!\}\ a +\!\!+ \mathit{encode}\{\!|\mathsf{b}|\!\}\ b\\
&\mathit{encode}\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (\mathit{Inl}\ a) &&= \mathit{encode}\{\!|\mathsf{a}|\!\}\ a\\
&\mathit{encode}\{\!|\mathsf{a} :\!+\!: \mathsf{b}|\!\}\ (\mathit{Inr}\ b) &&= \mathit{encode}\{\!|\mathsf{b}|\!\}\ b.
\end{aligned}$$

For Unit and String there is nothing to encode. The product case encodes the components of the product, and concatenates the results. The sum case strips of the *Inl* or *Inr* constructor, and encodes the argument.

The encoding happens in the constructor case of function *encode*. We use function *intinrange2bits* to calculate the bits for the position of the argument constructor in the constructor list, given the number of constructors of the data type currently in scope. The definition of *intinrange2bits* is omitted.

$$\begin{aligned}
&\mathit{encode}\{\!|\mathsf{Con}\ c\ \mathsf{a}|\!\}\ (\mathit{Con}\ a) =\\
&\qquad \mathit{intinrange2bits}\ (\mathit{constructorPosition}\ c) +\!\!+ \mathit{encode}\{\!|\mathsf{a}|\!\}\ a\\
&\mathit{intinrange2bits} :: (\mathsf{Int}, \mathsf{Int}) \to \mathsf{Bin}
\end{aligned}$$

We omit the definition of the function to decode a list of bits into a value of a data type. This function is the inverse of function *encode* defined in this section, and is very similar to the function *decodes* given in the first part of these lecture notes.

Compressing the Contents. Finally, it remains to compress the contents of an XML document. At the moment we use the Unix compress utility [52] to compress the strings obtained from the document.

3.2 Analysis

How does XCOMPREZ perform, and how does it compare with other XML compressors? The analysis given in this section is limited: XCOMPREZ is used as an example of a generic program, and not as the latest development in XML compression. Furthermore, we have not been able to obtain the executables or the source code of most of the existing XML compressors.

Existing XML Compressors. Structure-specific compression methods give much better compression results [4, 15, 14, 46] than conventional compression methods such as the Unix compress utility [52]. There exist many XML compressors; we know of XMLZip [12], XMill [36], ICT's XML-Xpress [25], Millau [16], XMLPPM [6], XGrind [48], and lossy XML compression [5]. We will not perform an exhaustive comparison between our compressor and these compressors, but we will briefly compare our compressor with XMill.

Compression Ratio. We have performed some initial tests comparing XCOMPREZ and XMill. The tests are not representative, and it is impossible to draw hard conclusions from the results. However, on our test examples XCOMPREZ is 40% to 50% better than XMill. We think this improvement in compression ratio is considerable. When we replace HaXml by a tool that generates a data type for a schema, we expect that we can achieve better compression ratios.

Code Size. With respect to code size, the difference between XMill and XCOMPREZ is dramatic: XMill is written in almost 20k lines of C++. The main functionality of XCOMPREZ is less than 300 lines of Generic Haskell code. Of course, for a fair comparison we have to add some of the HaXml code (which is a library distributed together with almost all compiler and interpreters for Haskell), the code for handling bits, and the code for implementing the as yet unimplemented features of XMill. We expect to be able implement all of XMill's features in about 20% of the code size of XMill.

Extensions of XCOMPREZ. A relatively simple way to improve XCOMPREZ it is to analyze some source files that are valid with respect to the DTD, count the number of occurrences of the different elements (constructors), and apply Huffman coding. We have implemented this rather simple extension [32].

XMill stores the strings in the elements in so-called containers. The standard approach in XMill is to use different containers for different elements, so that, for example, all authors are stored in the author container, all dates in the date container, etc. Since the strings in for example the container for dates are very similar, standard compression methods can compress the containers with a larger factor than the single file obtained by storing all strings that appear in the XML document. Again, it is easy to implement this feature as a generic program [32].

Finally, we have also used (Adaptive) Arithmetic Coding [3] to compress the constructors.

Specializing XCOMPREZ. XCOMPREZ can not only be used for compressing XML documents, but also for compressing values of arbitrary data types that have not necessarily been generated by DtdToHaskell.

Suppose we have a data type that contains a constructor *Age*, which takes an integer as argument and denotes the age of a human being. Since 128 is currently a safe upperbound for the age of a human being, it suffices to use 7 bits for the integer argument of *Age*. Suppose *compressAge* calculates these 7 bits from an age. Then we can reuse the definition of *encode* together with a constructor case [10] to define a function *specialEncode*.

$$
\begin{aligned}
&\mathit{specialEncode}\{\!|\mathsf{t} :: \kappa|\!\} && :: \mathsf{Encode}\{[\kappa]\}\ \mathsf{t}\\
&\mathit{specialEncode}\{\!|\mathbf{case}\ \mathit{Age}|\!\}\ (\mathit{Age}\ i) && = \mathit{compressAge}\ i\\
&\mathit{specialEncode}\{\!|\mathsf{a}|\!\} && = \mathit{encode}\{\!|\mathsf{a}|\!\}
\end{aligned}
$$

Function *specialEncode* is still a generic encoding function, but on values of the form *Age i* it uses a different compress function.

3.3 Conclusions

We have shown how to implement an XML compressor as a generic program. XCOMPREZ compresses better than for example XMill because it uses the information about an XML document present in a DTD.

There exist several other classes of XML tools that can be implemented as generic programs, and that would benefit from such an implementation. Examples of such tools are XML editors and XML databases [17]. The combination of HaXml and generic programming as in Generic Haskell is very useful for implementing the kind of XML tools for which DTDs play an important rôle. Using generic programming, such tools become easier to write, because a lot of the code pertaining to DTD handling and optimisation is obtained from the generic programming compiler, and the resulting tools are more effective, because they directly depend on the DTD. For example, a DTD-aware XML compressor, such as XCOMPREZ described in this paper, compresses considerably better than XML compressors that don't take the DTD into account, such as XMill. Furthermore, our compressor is much smaller than XMill.

Although we think Generic Haskell is very useful for developing DTD-aware XML tools, there are some features of XML tools that are difficult to express in Generic Haskell. Some of the functionality in the DOM, such as the methods `childNodes` and `firstChild` in the `Node` interface, is hard to express in a typed way. A flexible extension of type-indexed data types [23] might offer a solution to this problem. We believe that fusing HaXml, or a tool based on Schemas, with Generic Haskell, obtaining a 'domain-specific' language [8] for generic programming on DTDs or Schemas is a promising approach.

For tools that do not depend on a DTD we can use the untyped approach from HaXml to obtain a tool that works for any document. However, most of the advantages of generic programming no longer apply.

Exercise 3. Adapt the function *extract* such that it returns a list of containers, where a container is a list of strings. Return a (possibly empty) container for every constructor name.

Exercise 4. There might be many empty containers when using the approach from the previous exercise. Analyse a data type for occurrences of the type String under constructors. Use this analysis to only return containers for constructor names that might contain strings.

Exercise 5. Function *insert* takes the shape and the contents (a list of strings) of a value, and inserts the strings at the right positions in the shape. Define function *insert* as a a generic function with a kind-indexed type.

Exercise 6. Adapt the current version of XCOMPREZ such that it can use Huffman coding instead of the standard constructor encoding used in this section. Make sure other encodings can be used as well.

4 The Zipper

This section shows how to define a so-called zipper for an arbitrary data type. This is an advanced example demonstrating the full power of a type-indexed data type together with a number of generic functions working on it.

The zipper is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down in the tree. The zipper is used in tools where a user interactively manipulates trees, for instance, in editors for structured documents such as proofs or programs. For the following it is important to note that the focus of the zipper may only move to recursive components. Consider as an example the data type Tree:

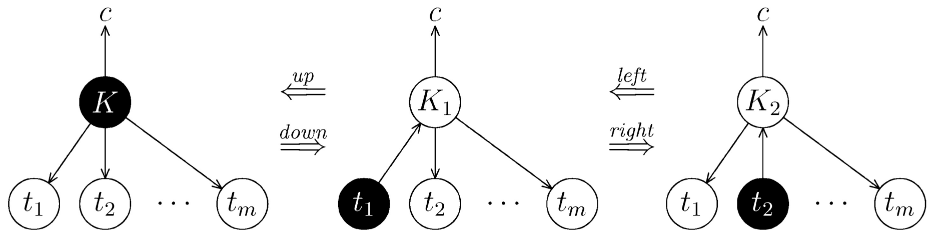
```
data Tree a b = Tip a | Node (Tree a b) b (Tree a b).
```

If the left subtree of a *Node* constructor is the current focus, moving right means moving to the right tree, not to the b-label. This implies that recursive positions in trees play an important rôle in the definition of a generic zipper data structure. To obtain access to these recursive positions, we have to be explicit about the fixed points in data type definitions. The zipper data structure is then defined by induction on the so-called pattern functor of a data type.

The tools in which the zipper is used, allow the user to repeatedly apply navigation or edit commands, and to update the focus accordingly. In this section we define a type-indexed data type for locations, which consist of a subtree (the focus) together with a context, and we define several navigation functions on locations.

4.1 The Basic Idea

The zipper is based on pointer reversal. If we follow a pointer to a subterm, the pointer is reversed to point from the subterm to its parent so that we can go up again later. A location is a pair (t, c) consisting of the current subterm t and a pointer c to its parent. The upward pointer corresponds to the *context* of the subterm. It can be represented as follows. For each constructor K that has m recursive subcomponents we introduce m context constructors $K_1, \ldots, K_m$. Now, consider the location $(K\ t_1\ t_2\ \ldots\ t_m, c)$. If we go down to t_1, we are left with the context $K\ \bullet\ t_2\ \ldots\ t_m$ and the old context c. To represent the combined context, we simply plug c into the hole to obtain $K_1\ c\ t_2\ \ldots\ t_m$. Thus, the new location is $(t_1, K_1\ c\ t_2\ \ldots\ t_m)$. The following picture illustrates the idea (the filled circle marks the current cursor position).



4.2 Data Types as Fixed Points of Pattern Functors

As mentioned above, in order to use the zipper, we have to be explicit about the fixed points in data type definitions. Therefore, we introduce the data type Fix, which is used to define a data type as a fixed point of a pattern functor. The pattern functor makes the recursion explicit in a data type.

$$\mathbf{newtype}\ \mathsf{Fix\ f} = \mathit{In}\{\ \mathit{out} :: \mathsf{f\ (Fix\ f)}\}$$

This is a labelled variant of the data type Fix defined in Section 1.2 of the first part of these lecture notes. For example, the data types of natural numbers and bushes can be defined using explicit fixed points as follows:

```
data NatF a  = ZeroF | SuccF a
type Nat     = Fix NatF
data BushF a = LeafF Char | BinF a a
type Bush    = Fix BushF.
```

It is easy to convert between data types defined as fixed points and the original data type definitions of natural numbers and bushes. However, nested data types and mutually recursive data types cannot be defined in terms of this particular definition of Fix.

4.3 Type Indices of Higher Kinds

The types that occur in the indices of a generic function have kind $\star$ as their base kind. For example, Int, Char and Unit are all of kind $\star$, and `:+:` and `:*:`

have kind $\star \to \star \to \star$. In this section we are going to define generic functions which have $\star \to \star$ as their base kind. We need slightly different type indices for generic functions operating on types of kind $\star \to \star$:

$$\begin{aligned}
&\mathsf{K\ t} &&= \Lambda \mathsf{a}\,.\,\mathsf{t}\\
&\mathsf{f1\ {:}{+}{:}\ f2} &&= \Lambda \mathsf{a}\,.\,\mathsf{f1\ a\ {:}{+}{:}\ f2\ a}\\
&\mathsf{f1\ {:}{*}{:}\ f2} &&= \Lambda \mathsf{a}\,.\,\mathsf{f1\ a\ {:}{*}{:}\ f2\ a}\\
&\mathsf{Con}\ c\ \mathsf{f} &&= \Lambda \mathsf{a}\,.\,\mathsf{Con}\ c\ (\mathsf{f\ a})\\
&\mathsf{Id} &&= \Lambda \mathsf{a}\,.\,\mathsf{a}.
\end{aligned}$$

We have the constant functor K, which lifts a type of kind $\star$ to kind $\star \to \star$. We will need K Unit as well as K Char (or more general, K t for all primitive types). We overload `:+:`, `:*:`, and Con, to be lifted versions of their previously defined counterparts. The only new type index in this set of indices of kind $\star \to \star$ is the identity functor Id. Hinze [20] shows that these types are the normal forms of types of kind $\star \to \star$.

4.4 Locations

A location is a subtree, together with a context, which encodes the path from the top of the original tree to the selected subtree. The type-indexed data type Loc returns a type for locations given an argument pattern functor.

$$\begin{aligned}
&\mathsf{Loc}\{\!|\mathsf{f} :: \star \to \star|\!\} &&:: \quad \star\\
&\mathsf{Loc}\{\!|\mathsf{f}|\!\} &&= \quad (\mathsf{Fix\ f}, \mathsf{Context}\{\!|\mathsf{f}|\!\}\ (\mathsf{Fix\ f}))\\
&\mathsf{Context}\{\!|\mathsf{f} :: \star \to \star|\!\} &&:: \quad \star \to \star\\
&\mathsf{Context}\{\!|\mathsf{f}|\!\}\ \mathsf{r} &&= \quad \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{Ctx}\{\!|\mathsf{f}|\!\}\ \mathsf{r}))\\
&\mathbf{data}\ \mathsf{LMaybe\ f\ a} &&= \quad \mathit{LNothing} \mid \mathit{LJust}\ (\mathsf{f\ a}),
\end{aligned}$$

where LMaybe is the lifted version of Maybe. The type Loc is defined in terms of Context, which constructs the context parameterized by the original tree type. The Context of a value is either empty (represented by *LNothing* in the LMaybe type), or it is a path from the root down into the tree. Such a path is constructed by means of the argument type of LMaybe: the type-indexed data type Ctx. The type-indexed data type Ctx is defined by induction on the pattern functor of the original data type. It can be seen as the *derivative* (as in calculus) of the pattern functor f [39, 1]. If the derivative of f is denoted by f′, we have

$$\begin{aligned}
\mathsf{const}' &= 0\\
(\mathsf{f}+\mathsf{g})' &= \mathsf{f}'+\mathsf{g}'\\
(\mathsf{f}\times\mathsf{g})' &= \mathsf{f}'\times\mathsf{g}+\mathsf{f}\times\mathsf{g}'.
\end{aligned}$$

It follows that in the definition of Ctx we will also need access to the type arguments themselves on the right-hand side of the definition.

$$
\begin{array}{lcl}
\mathsf{Ctx}\{\!|\mathsf{f} :: \star \to \star|\!\} & :: & \star \to \star \to \star \\
\mathsf{Ctx}\{\!|\mathsf{Id}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{c} \\
\mathsf{Ctx}\{\!|\mathsf{K\ Unit}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Void} \\
\mathsf{Ctx}\{\!|\mathsf{K\ Char}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Void} \\
\mathsf{Ctx}\{\!|\mathsf{f1} :+: \mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} & = & \mathsf{Ctx}\{\!|\mathsf{f1}|\!\}\ \mathsf{r}\ \mathsf{c} :+: \mathsf{Ctx}\{\!|\mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} \\
\mathsf{Ctx}\{\!|\mathsf{f1} :*: \mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c} & = & (\mathsf{Ctx}\{\!|\mathsf{f1}|\!\}\ \mathsf{r}\ \mathsf{c} :*: \mathsf{f2}\ \mathsf{r}) :+: (\mathsf{f1}\ \mathsf{r} :*: \mathsf{Ctx}\{\!|\mathsf{f2}|\!\}\ \mathsf{r}\ \mathsf{c})
\end{array}
$$

This definition can be understood as follows. Since it is not possible to descend into a constant, the constant cases do not contribute to the result type, which is denoted by the 'empty type' Void, a type without values. The Id case denotes a recursive component, in which it is possible to descend. Hence it may occur in a context. Descending in a value of a sum type follows the structure of the input value. Finally, there are two ways to descend in a product: descending left, adding the contents to the right of the node to the context, or descending right, adding the contents to the left of the node to the context.

For example, for natural numbers with pattern functor K Unit :+: Id, and for trees of type Bush with pattern functor BushF, which can be represented by K Char :+: (Id :*: Id) we obtain

$$
\begin{array}{ll}
\mathsf{Context}\{\!|\mathsf{K\ Unit} :+: \mathsf{Id}|\!\}\ \mathsf{r} & = \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{NatC}\ \mathsf{r})) \\
\mathsf{Context}\{\!|\mathsf{K\ Char} :+: \mathsf{Id} :*: \mathsf{Id}|\!\}\ \mathsf{r} & = \mathsf{Fix}\ (\mathsf{LMaybe}\ (\mathsf{BushC}\ \mathsf{r})) \\
\mathbf{data}\ \mathsf{NatC}\ \mathsf{r}\ \mathsf{c} & = \mathit{ZeroC}\ \mathsf{Void} \mid \mathit{SuccC}\ \mathsf{c} \\
\mathbf{data}\ \mathsf{BushC}\ \mathsf{r}\ \mathsf{c} & = \mathit{LeafC}\ \mathsf{Void} \mid \mathit{BinCL}\ (\mathsf{c}, \mathsf{r}) \mid \mathit{BinCR}\ (\mathsf{r}, \mathsf{c}).
\end{array}
$$

The context of a natural number is isomorphic to a natural number (the context of m in n is $n - m$), and the context of a Bush applied to the data type Bush itself is isomorphic to the type Context_Bush introduced in Section 1.

McBride [39, 1] also defines a type-indexed zipper data type. His zipper slightly deviates from Huet's and our zipper: the navigation functions on McBride's zipper are not constant time anymore. The observation that the Context of a data type is its derivative (as in calculus) is due to McBride.

4.5 Navigation Functions

We define generic functions on the type-indexed data types Loc, Context, and Ctx for navigating through a tree. All of these functions act on locations. These are the basic functions for the zipper.

Function down. The function *down* is a generic function that moves down to the leftmost recursive child of the current node, if such a child exists. Otherwise, if the current node is a leaf node, then *down* returns the location unchanged.

$$\mathit{down}\{\!|\mathsf{f} :: \star \to \star|\!\} :: \mathsf{Loc}\{\!|\mathsf{f}|\!\} \to \mathsf{Loc}\{\!|\mathsf{f}|\!\}$$

The instantiation of *down* to the data type Bush has been given in Section 1. The function *down* satisfies the following property:

$$\forall m\, .\, \mathit{down}\{\!|\mathsf{f}|\!\}\ m \neq m \implies (\mathit{up}\{\!|\mathsf{f}|\!\} \cdot \mathit{down}\{\!|\mathsf{f}|\!\})\ m = m,$$

where the function *up* goes up in a tree. So first going down the tree and then up again is the identity function on locations in which it is possible to go down.

Since *down* moves down to the leftmost recursive child of the current node, the inverse equality $\mathit{down}\{|\mathsf{f}|\} \cdot \mathit{up}\{|\mathsf{f}|\} = \mathit{id}$ does not hold in general. However, there does exist a natural number n such that

$$\forall m \,.\, \mathit{up}\{|\mathsf{f}|\}\ m \neq m \implies (\mathit{right}\{|\mathsf{f}|\}^n \cdot \mathit{down}\{|\mathsf{f}|\} \cdot \mathit{up}\{|\mathsf{f}|\})\ m = m,$$

where the function *right* goes right in a tree. These properties do not completely specify function *down*. The other properties it should satisfy are that the selected subtree of $\mathit{down}\{|\mathsf{f}|\}\ m$ is the leftmost tree-child of the selected subtree of m, and the context of $\mathit{down}\{|\mathsf{f}|\}\ m$ is the context of m extended with all but the leftmost tree-child of m.

The function *down* is defined as follows.

```
down{|f|} (t, c) = case first{|f|} (out t) c of
                     Just (t', c') → (t', In (LJust c'))
                     Nothing → (t, c)
```

To find the leftmost recursive child, we have to pattern match on the pattern functor f, and find the first occurrence of Id. The helper function *first* is a generic function that possibly returns the leftmost recursive child of a node, together with the context (a value of type Ctx{|f|} c t) of the selected child. The function *down* then turns this context into a value of type Context by inserting it in the right ('non-top') component of a sum by means of *LJust*, and applying the fixed point constructor *In* to it.

```
first{|f :: ⋆ → ⋆|}          :: ∀c t . f t → c → Maybe (t, Ctx{|f|} c t)
first{|Id|} t c              = return (t, c)
first{|K Unit|} t c          = Nothing
first{|K Char|} t c          = Nothing
first{|f1 :+: f2|} (Inl x) c = do {(t, cx) ← first{|f1|} x c; return (t, Inl cx)}
first{|f1 :+: f2|} (Inr y) c = do {(t, cy) ← first{|f2|} y c; return (t, Inr cy)}
first{|f1 :*: f2|} (x :*: y) c = do {(t, cx) ← first{|f1|} x c
                                      ; return (t, Inl (cx, y))}
                              ++ do {(t, cy) ← first{|f2|} y c
                                      ; return (t, Inr (x, cy))}
```

Here, *return* is obtained from the Maybe monad, and the operator (++) is the standard monadic plus, called *mplus* in Haskell, given by

```
(++)          :: ∀a . Maybe a → Maybe a → Maybe a
Nothing ++ m = m
Just a ++ m  = Just a.
```

The function *first* returns the value and the context at the leftmost Id position. So in the product case, it first tries the left component, and only if it fails, it tries the right component.

The definitions of functions *up*, *right* and *left* are not as simple as the definition of *down*, since they are defined by pattern matching on the context instead of on the tree itself. We will just define functions *up* and *right*, and leave function *left* as an exercise.

Function up. The function *up* moves up to the parent of the current node, if the current node is not the top node.

```
up{|f :: ⋆ → ⋆|} :: Loc{|f|} → Loc{|f|}
up{|f|} (t, c)   = case out c of
                     LNothing → (t, c)
                     LJust c′ → do {ft ← insert{|f|} c′ t;
                                    c″ ← extract{|f|} c′;
                                    return (In ft, c″)}
```

Remember that *LNothing* denotes the empty top context. The navigation function *up* uses two helper functions: *insert* and *extract*. The latter returns the context of the parent of the current node. Each element of type Ctx{|f|} c t has at most one c component (by an easy inductive argument), which marks the context of the parent of the current node. The generic function *extract* extracts this context.

```
extract{|f :: ⋆ → ⋆|}               :: ∀c t . Ctx{|f|} c t → Maybe c
extract{|Id|} c                      = return c
extract{|K Unit|} c                  = Nothing
extract{|K Char|} c                  = Nothing
extract{|f1 :+: f2|} (Inl cx)        = extract{|f1|} cx
extract{|f1 :+: f2|} (Inr cy)        = extract{|f2|} cy
extract{|f1 :*: f2|} (Inl (cx, y))   = extract{|f1|} cx
extract{|f1 :*: f2|} (Inr (x, cy))   = extract{|f2|} cy
```

The function *extract* is polymorphic in c and in t.

Function *insert* takes a context and a tree, and inserts the tree in the current focus of the context, effectively turning a context into a tree.

```
insert{|f :: ⋆ → ⋆|}                  :: ∀c t . Ctx{|f|} c t → t → Maybe (f t)
insert{|Id|} c t                       = return t
insert{|K Unit|} c t                   = Nothing
insert{|K Char|} c t                   = Nothing
insert{|f1 :+: f2|} (Inl cx) t         = do {x ← insert{|f1|} cx t; return (Inl x)}
insert{|f1 :+: f2|} (Inr cy) t         = do {y ← insert{|f2|} cy t; return (Inr y)}
insert{|f1 :*: f2|} (Inl (cx, y)) t    = do {x ← insert{|f1|} cx t; return (x, y)}
insert{|f1 :*: f2|} (Inr (x, cy)) t    = do {y ← insert{|f2|} cy t; return (x, y)}
```

Note that the extraction and insertion is happening in the identity case Id; the other cases only pass on the results.

Since $up\{|f|\} \cdot down\{|f|\} = id$ on locations in which it is possible to go down, we expect similar equalities for the functions *first*, *extract*, and *insert*. We have that the following computation

```
do { (t, c') ← first{|f|} ft c;
     c'' ← extract{|f|} c';
     ft' ← insert{|f|} c' t        ;
     return (c == c'' ∧ ft == ft' ) },
```

returns *true* on locations in which it is possible to go down.

Function right. The function *right* moves the focus to the next (right) sibling in a tree, if it exists. The context is moved accordingly. The instance of *right* on the data type Bush has been given in Section 1. The function *right* satisfies the following property:

$$\forall m\,.\,\mathit{right}\{|\mathsf{f}|\}\ m \neq m \quad\Longrightarrow\quad (\mathit{left}\{|\mathsf{f}|\} \cdot \mathit{right}\{|\mathsf{f}|\})\ m = m,$$

that is, first going right in the tree and then left again is the identity function on locations in which it is possible to go to the right. Of course, the dual equality holds on locations in which it is possible to go to the left. Furthermore, the selected subtree of *right*{|f|} *m* is the sibling to the right of the selected subtree of *m*, and the context of *right*{|f|} *m* is the context of *m* in which the context is replaced by the selected subtree of *m*, and the first subtree to the right of the context of *m* is replaced by the context of *m*.

Function *right* is defined by pattern matching on the context. It is impossible to go to the right at the top of a tree. Otherwise, we try to find the right sibling of the current focus.

```
right{|f :: ⋆ → ⋆|} :: Loc{|f|} → Loc{|f|}
right{|f|} (t, c)   =  case out c of
                         LNothing → (t, c)
                         LJust c' → case next{|f|} t c' of
                                      Just (t', c'') → (t', In (LJust c''))
                                      Nothing → (t, c)
```

The helper function *next* is a generic function that returns the first location that has the recursive value to the right of the selected value as its focus. Just as there exists a function *left* such that *left*{|f|} · *right*{|f|} = *id* (on locations in which it is possible to go to the right), there exists a function *previous*, such that

```
do { (t', c') ← next{|f|} t c ;
     (t'', c'') ← previous{|f|} t' c';
     return (c == c'' ∧ t == t'')},
```

returns *true* (on locations in which it is possible to go to the right). We will define function *next*, and omit the definition of function *previous*.

```
$next\{|\mathsf{f} :: \star \to \star|\} :: \forall \mathsf{c}\ \mathsf{t} . \mathsf{t} \to \mathsf{Ctx}\{|\mathsf{f}|\}\ \mathsf{c}\ \mathsf{t} \to \mathsf{Maybe}\ (\mathsf{t}, \mathsf{Ctx}\{|\mathsf{f}|\}\ \mathsf{c}\ \mathsf{t})$
$next\{|\mathsf{Id}|\}\ t\ c$                      $= Nothing$
$next\{|\mathsf{K\ Unit}|\}\ t\ c$                $= Nothing$
$next\{|\mathsf{K\ Char}|\}\ t\ c$                $= Nothing$
$next\{|\mathsf{f1} :+: \mathsf{f2}|\}\ t\ (Inl\ cx)$
  $= \mathbf{do}\ \{ (t', cx') \leftarrow next\{|\mathsf{f1}|\}\ t\ cx; return\ (t', Inl\ cx') \}$
$next\{|\mathsf{f1} :+: \mathsf{f2}|\}\ t\ (Inr\ cy)$
  $= \mathbf{do}\ \{ (t', cy') \leftarrow next\{|\mathsf{f2}|\}\ t\ cy; return\ (t', Inr\ cy') \}$
$next\{|\mathsf{f1} :*: \mathsf{f2}|\}\ t\ (Inl\ (cx, y)\ )$
  $= \mathbf{do}\ \{ (t', cx') \leftarrow next\{|\mathsf{f1}|\}\ t\ cx; return\ (t', Inl\ (cx', y)) \}$
          $+\!\!+ \mathbf{do}\ \{ c \leftarrow extract\{|\mathsf{f1}|\}\ cx;$
                 $x \leftarrow insert\{|\mathsf{f1}|\}\ cx\ t;$
                 $(t', cy) \leftarrow first\{|\mathsf{f2}|\}\ y\ c;$
                 $return\ (t', Inr\ (x, cy)) \}$
$next\{|\mathsf{f1} :*: \mathsf{f2}|\}\ t\ (Inr\ (x, cy))$
  $= \mathbf{do}\ \{ (t', cy') \leftarrow next\{|\mathsf{f2}|\}\ t\ cy; return\ (t', Inr\ (x, cy')) \}$
```

The first three lines in this definition show that it is impossible to go to the right in an identity or constant context. If the context argument is a value of a sum, we select the next element in the appropriate component of the sum. The product case is the most interesting one. If the context is in the right component of a pair, *next* returns the next value of that context, properly combined with the left component of the tuple. On the other hand, if the context is in the left component of a pair, the next value may be either in that left component (the context), or it may be in the right component (the value). If the next value is in the left component, it is returned by the first line in the definition of the product case. If it is not, *next* extracts the context c (the context of the parent) from the left context cx, it inserts the given value in the context cx giving a 'tree' value x, and selects the first component in the right component of the pair, using the extracted context c for the new context. The new context that is thus obtained is combined with x into a context for the selected tree.

Exercise 7. Define the function *left*:

$$left\{|\mathsf{f} :: \star \to \star|\} :: \mathsf{Loc}\{|\mathsf{f}|\} \to \mathsf{Loc}\{|\mathsf{f}|\}.$$

Exercise 8. If you don't want to use the zipper, you can alternatively keep track of the path to the current focus. Suppose we want to use the path to determine the name of the top constructor of the current focus in a value of a data type. The path determines which child of a value is selected. Since the products used in our representations of data types are binary, a path has the following structure:

```
$\mathbf{data}\ \mathsf{Dir}\ \ \ = L \mid R$
$\mathbf{type}\ \mathsf{Path} = [\mathsf{Dir}].$
```

The to be defined function *selectCon* takes a value of a data type and a path, and returns the constructor name at the position denoted by the path. For example,

$$\begin{array}{l}
\textbf{data}\ \mathsf{List}\ = \mathit{Nil} \mid \mathit{Cons}\ \mathsf{Char}\ \mathsf{List} \\
\mathit{selectCon}\{\!|\mathsf{List}|\!\}\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ (\mathit{Cons}\ 6\ \mathit{Nil})))\ [R, R, R] \\
\Longrightarrow \texttt{"Nil"} \\
\textbf{data}\ \mathsf{Tree} = \mathit{Leaf}\ \mathsf{Int} \mid \mathit{Node}\ \mathsf{Tree}\ \mathsf{Int}\ \mathsf{Tree} \\
\mathit{selectCon}\{\!|\mathsf{Tree}|\!\}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R] \\
\Longrightarrow \texttt{"Node"} \\
\mathit{selectCon}\{\!|\mathsf{Tree}|\!\}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ 3\ (\mathit{Node}\ (\mathit{Leaf}\ 2)\ 1\ (\mathit{Leaf}\ 5)))\ [R, R, L] \\
\Longrightarrow \texttt{"Leaf"}.
\end{array}$$

Define the generic function *selectCon*, together with its kind-indexed type.

Exercise 9. Define the function *left*, which takes a location, and returns the location to the left of the argument location, if possible.

Exercise 10. For several applications we have to extend a data type such that it is possible to represent a place holder. For example, from the data type Tree defined by

$$\textbf{data}\ \mathsf{Tree\ a\ b} = \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{Tree\ a\ b})\ \mathsf{b}\ (\mathsf{Tree\ a\ b}),$$

we would like to obtain a type isomorphic to the following type:

$$\textbf{data}\ \mathsf{HoleTree\ a\ b} = \mathit{Hole} \mid \mathit{Tip}\ \mathsf{a} \mid \mathit{Node}\ (\mathsf{HoleTree\ a\ b})\ \mathsf{b}\ (\mathsf{HoleTree\ a\ b}).$$

– Define a type-indexed data type Hole that takes a data type and returns a data type in which also holes can be specified. Also give the kind-indexed kind of this type-indexed data type. (The kind-indexed kind cannot and does not have to be defined in Generic Haskell though.)
– Define a generic function *toHole* which translates a value of a data type t to a value of the data type Hole{|t|}, and a function *fromHole* that does the inverse for values that do not contain holes anymore:

$$\begin{array}{lll}
\mathit{toHole}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{ToHole}\{\![\kappa]\!\}\ \mathsf{t} \\
\mathit{fromHole}\{\!|\mathsf{t} :: \kappa|\!\} & :: & \mathsf{FromHole}\{\![\kappa]\!\}\ \mathsf{t}
\end{array}$$

$$\begin{array}{ll}
\textbf{type}\ \mathsf{ToHole}\{\![\star]\!\}\ \mathsf{t} & = \mathsf{t} \rightarrow \mathsf{Hole}\{\!|\mathsf{t}|\!\} \\
\textbf{type}\ \mathsf{FromHole}\{\![\star]\!\}\ \mathsf{t} & = \mathsf{Hole}\{\!|\mathsf{t}|\!\} \rightarrow \mathsf{t}.
\end{array}$$

5 Conclusions

We have developed three advanced applications in Generic Haskell. In these examples we use, besides generic functions with kind-indexed kinds, type-indexed data types, dependencies between and generic abstractions of generic functions, and default and constructor cases. Some of the latest developments of Generic Haskell have been guided by requirements from these applications.

We hope to develop more applications using Generic Haskell in the future, both to develop the theory and the language. Current candidate applications are more XML tools and editors.

Acknowledgements

Andres Löh implemented Generic Haskell and the zipper example, and contributed to almost all other examples. Paul Hagg contributed to the implementation of the XML compressor. Dave Clarke, Andres Löh, Ralf Lämmel, Doaitse Swierstra and Jan de Wit commented on or contributed to (parts of) previous versions of this paper.

References

1. Michael Abott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, TLCA 2003*, 2003. To appear.
2. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.
3. Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International Summer School, Oxford, UK*, volume 2638 of *LNCS*. Springer-Verlag, 2003. To appear.
4. Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.
5. Mario Cannataro, Gianluca Carelli, Andrea Pugliese, and Domenico Sacca. Semantic lossy compression of XML data. In *Knowledge Representation Meets Databases*, 2001.
6. James Cheney. Compressing xml with multiplexed hierarchical models. In *Proceedings of the 2001 IEEE Data Compression Conference, DCC'01*, pages 163–172, 2001.
7. Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using Quickcheck and Hat. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional programming, 4th International Summer School, Oxford, UK*, volume 2638 of *LNCS*. Springer-Verlag, 2003. To appear.
8. Dave Clarke. Towards GH(XML). Talk at the Generic Haskell meeting, see `http://www.generic-haskell.org/talks.html`, 2001.
9. Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001. Also available from `http://www.generic-haskell.org/`.
10. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, volume 243 of *IFIP*, pages 21–48. Kluwer Academic Publishers, January 2003.
11. Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(3):381–418, September 1995.
12. XMLSolutions Corporation. XMLZip. Available from `http://www.xmlzip.com/`, 1999.
13. René de la Briandais. File searching using variable length keys. In *Proc. Western Joint Computer Conference*, volume 15, pages 295–298. AFIPS Press, 1959.

14. William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.
15. Michael Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 263–276. Springer-Verlag: Heidelberg, Germany, 1997.
16. Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. In *IEEE International Conference on Multimedia and Expo (I) 2000*, pages 747–765, 2000.
17. Paul Hagg. A framework for developing generic XML Tools. Master's thesis, Department of Information and Computing Sciences, Utrecht University, 2002.
18. Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
19. Ralf Hinze. *Generic Programs and Proofs.* 2000. Habilitationsschrift, Bonn University.
20. Ralf Hinze. A new approach to generic functional programming. In *Conference Record of POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 2000.
21. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2-3):129–159, 2002.
22. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory, 2003. To appear.
23. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference, MPC'02*, volume 2386 of *LNCS*, pages 148–174, 2002.
24. Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
25. INC Intelligent Compression Technologies. XML-Xpress. Whitepaper available from `http://www.ictcompress.com/products_xmlxpress.html`, 2001.
26. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.
27. Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2001.
28. Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.
29. Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In J. Jeuring, editor, *Workshop on Generic Programming 2000, Ponte de Lima, Portugal, July 2000*, pages 33–45, 2000. Utrecht Technical Report UU-CS-2000-19.
30. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
31. J. Jeuring. Polytypic pattern matching. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 238–248. ACM Press, 1995.
32. Johan Jeuring and Paul Hagg. XComprez. Available from `http://www.generic-haskell.org/xmltools/XComprez/`, 2002.
33. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley Publishing Company, 2nd edition, 1998.
34. Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: Generic automated software testing. Submitted for publication, 2002.

35. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, 2003.
36. Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, 2000.
37. Andres Löh, Dave Clarke, and Johan Jeuring. Generic Haskell, naturally: The language and its type system. In preparation, 2003.
38. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
39. Connor McBride. The derivative of a regular type is its type of one-hole contexts. Unpublished manuscript, 2001.
40. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.
41. Lambert Meertens. Functor pulling. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
42. E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *FPCA'91: Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, 1991.
43. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.
44. Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, pages 77–86, 1998.
45. Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.
46. C.H. Stork, V. V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Information and Computer Science, University of California, Irvine, 2000.
47. Axel Thue. Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Skrifter udgivne af Videnskaps-Selskabet i Christiania, Mathematisk-Naturvidenskabelig Klasse*, 1:1–67, 1912. Reprinted in Thue's "Selected Mathematical Papers" (Oslo: Universitetsforlaget, 1977), 413–477.
48. Pankaj Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, 2002.
49. W3C. XML 1.0. Available from `http://www.w3.org/XML/`, 1998.
50. C.P. Wadsworth. Recursive type operators which are more than type schemes. *Bulletin of the EATCS*, 8:87–88, 1979. Abstract of a talk given at the 2nd International Workshop on the Semantics of Programming Languages, Bad Honnef, Germany, 19–23 March 1979.
51. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159, 1999.
52. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Generic Properties of Datatypes

Roland Backhouse [1] and Paul Hoogendijk [2]

[1] School of Computer Science and Information Technology
University of Nottingham, Nottingham NG8 1BB, UK
rcb@cs.nott.ac.uk

[2] Philips Research, Prof. Holstlaan 4, 5655 AA Eindhoven, The Netherlands
Paul.Hoogendijk@philips.com

Abstract. Generic programming adds a new dimension to the parametrisation of programs by allowing programs to be dependent on the structure of the data that they manipulate. Apart from the practical advantages of improved productivity that this offers, a major potential advantage is a substantial reduction in the burden of proof – by making programs more general we can also make them more robust and more reliable. These lectures discuss a theory of datatypes based on the algebra of relations which forms a basis for understanding datatype-generic programs. We review the notion of parametric polymorphism much exploited in conventional functional programming languages and show how it is extended to the higher-order notions of polymorphism relevant to generic programming.

1 Introduction

Imagine that you bought a processor designed to perform floating-point calculations with the utmost efficiency but later you discovered that it would not work correctly for all input values. You would, of course, demand your money back — even if the manufacturer tried to claim that the occasions on which the processor would fail were very few and far between. Now imagine that you were offered some software that was described as "generic" but came with the caveat that it had only been tried on a few instances and the extent of its "genericity" could not be formulated precisely, let alone guaranteed. Would you buy it, or would you prefer to wait and let others act as the guinea pigs?

"Generic programming" is important to software designers as a way of improving programmer productivity. But "generic" programs can only have value if the full extent of their "genericity" can be described clearly and precisely, and if the programs themselves can be validated against their specifications throughout the full range of their applicability.

These lectures are about a novel parameterisation mechanism that allows programs to be dependent on the structure of the data that they manipulate, so-called "datatype genericity". The focus is on defining clearly and precisely what the word "generic" really means. The (potential) benefit is a substantial reduction in the burden of proof.

In section 2, we review the notion of "parametric polymorphism" in the context of functional programming languages. This is a form of genericity which has long since proved its worth. For us, the important fact is that the notion can be given a precise definition, thus allowing one to formally verify whether or not a given function complies with the definition.

Section 3 introduces the notion of "commuting" two datatypes. This problem helps to illustrate the concepts introduced in the later sections. We seek a non-operational specification based on fundamental building blocks for a generic theory of datatypes.

Defining parametric polymorphism involves extending the notion of "mapping" functions over a datatype to mapping *relations* over a datatype. (This is unavoidably the case even in the context of functional programming.) Section 4 prepares the way for later sections by introducing the fundamentals of a relational theory of datatypes. For brevity and effective, calculational reasoning we use "point-free" relation algebra. The most important concept introduced in this section is that of a "relator" — essentially a datatype. Other, more well-known concepts like weakest liberal precondition, are formulated in a point-free style.

The "membership" relation and the associated "fan" of a relator are introduced in section 5. We argue that a datatype is a relator with membership. The introduction of a membership relation allows us to give greater insight into what it means for a polymorphic function to be a natural transformation. It also helps to distinguish between "proper" natural transformations and "lax" natural transformation.

In sections 6 and 7, we return to the issue of specifying when datatypes commute. This is formulated as the combination of two parametricity properties (one of which is high-level) and a homomorphism property (with respect to pointwise definition of relators).

2 Theorems for Free

In Pascal, it is necessary to define a different length function for lists of different types. For example, two different functions are needed to determine the length of a list of integers and to determine the length of a list of characters. This is a major inconvenience because, of course, the code for the two functions is identical; only their declarations are different. The justification given, at the time Pascal was designed, was that the increased redundancy facilitated type checking, thus assisting with the early detection of programming errors. In typed functional programming languages, beginning with ML [9, 10], only one so-called "polymorphic" function is needed. Formally, the length function is a function with type $\mathbb{N} \leftarrow \mathsf{List}.\alpha$ for all instances of the type parameter α. That is, whatever the type A, there is an instance of the function *length* that maps a list of A's to a natural number.

A polymorphic function is said to be "parametric" if its behaviour does not depend on the type at which it is instantiated [13]. That the length function on lists is parametric can be expressed formally. Suppose a function is applied to each element of a list —the function may, for example, map each character to a number— . Clearly, this will have no effect on the length of the list. Formally[1], we have, for all types A and B and all functions f of type $A \leftarrow B$,

$$length_A \circ \mathsf{List}.f \quad = \quad length_B \quad .$$

Note that this equation does not characterise the length function. Indeed, post-composing $length$ with any function g from natural numbers to natural numbers, we get a function that satisfies an equation of the same form. For example, let sq denote the function that squares a number. Then

$$(sq \circ length_A) \circ \mathsf{List}.f \quad = \quad sq \circ length_B \quad .$$

Also precomposing the length function with any function that maps a list to list, possibly duplicating or omitting elements of the list, but not performing any computation dependent on the value of the list element, also satisfies an equation of the same form. For example, suppose $copycat$ is a (polymorphic) function that maps a list to the list concatenated with itself. (So, for example, $copycat.$[0,1,2] equals [0,1,2,0,1,2] .) Then

$$(length_A \circ copycat_A) \circ \mathsf{List}.f \quad = \quad length_B \circ copycat_B \quad .$$

What is common to the functions $length$, $sq \circ length$ and $length \circ copycat$ is that they all have type $\mathbb{N} \leftarrow \mathsf{List}.\alpha$, for all instances of the type parameter α . Shortly, we define "parametric polymorphism" precisely. These three functions exemplify one instance of the definition. Specifically, g has so-called "polymorphic" type $\langle\forall\alpha :: \mathbb{N} \leftarrow \mathsf{List}.\alpha\rangle$ if for each type A there is a function g_A of type $\mathbb{N} \leftarrow \mathsf{List}.A$; g is so-called "parametrically" polymorphic if it has the property that, for all functions f of type $A \leftarrow B$,

$$g_A \circ \mathsf{List}.f \quad = \quad g_B \quad .$$

In general, a function will be called "parametrically polymorphic" if we can deduce a general equational property of the function from information about the type of the function alone.

A yet simpler example of a parametrically polymorphic function is the function fst that takes a pair of values and returns the first element of the pair. Using $\alpha{\times}\beta$ to denote the set of all pairs with first component of type α and second component of type β , the function fst has type $\langle\forall\ \alpha,\beta :: \alpha \leftarrow \alpha{\times}\beta\rangle$. Now

[1] Function composition is denoted by an infix " $\circ$ " symbol. Function application is denoted by an infix " . " symbol. By definition, $(f{\circ}g).x = f.(g.x)$. This is often called "backward composition" and " $\circ$ " is pronounced "after" (because f is executed "after" g). Because of this choice of order of evaluation, it is preferable to use a backward pointing arrow to indicate function types. The rule is that if f has type $A \leftarrow B$ and g has type $B \leftarrow C$ then $f{\circ}g$ has type $A \leftarrow C$. Functional programmers know the function " List " better by the name " `map` ". As we shall see, it is beneficial to give the type constructor and the associated " `map` " function the same name.

suppose f and g are functions of arbitrary type, and suppose $f{\times}g$ denotes the function that maps a pair of values (x,y) (of appropriate type) to the pair $(f.x\ ,\ g.y)$. Then, clearly,

$$\mathsf{fst} \circ f{\times}g \quad = \quad f \circ \mathsf{fst} \quad .$$

(Strictly, we should add type information by way of subscripts on the two occurrences of fst, just as we did for $length$. Later this type information will prove important, but for now we omit it from time to time.)

The simplest example of a parametrically polymorphic function is the identity function. For each type A there is an identity function id_A on A (the function such that, for all x of type A, $\mathsf{id}_A.x = x$.) Clearly, for all functions f of type $A \leftarrow B$,

$$\mathsf{id}_A \circ f \quad = \quad f \circ \mathsf{id}_B \quad .$$

What these examples illustrate is that a function is parametrically polymorphic if the function enjoys an algebraic property in common with all functions having the same type.

The key to a formal definition of "parametric polymorphism" is, in Wadler's words, "that types may be read as relations". This involves extending each type constructor —a function from types to types— to a relation constructor —a function from relations to relations— . The relations may, in fact, have arbitrary arity but we show how the extension is done just for binary relations.

For concreteness, we consider a language of types that comprises only function types and cartesian product. (In later sections we extend the language of types to include dataypes like List .) Specifically, we consider the type expressions defined by the following grammar:

$$Exp \quad ::= \quad Exp \times Exp \quad | \quad Exp \leftarrow Exp \quad | \quad Const \quad | \quad Var \quad .$$

Here, $Const$ denotes a set of constant types, like $\mathbb{N}$ (the natural numbers) and $\mathbb{Z}$ (the integers). Var denotes a set of type variables. We use Greek letters to denote type variables.

The syntactic ambiguity in the grammar is resolved by assuming that $\times$ has precedence over $\leftarrow$, and both are left associative. For example,

$$\mathbb{N} \;\leftarrow\; \mathbb{N} \;\leftarrow\; \alpha \times \alpha \times \beta$$

means

$$(\mathbb{N} \leftarrow \mathbb{N}) \leftarrow ((\alpha \times \alpha) \times \beta) \quad .$$

The inclusion of variables means that type expressions denote functions from types to types; we get a type for each instantiation of the type variables to types. The notation $T.A$ will be used to denote the type obtained by instantiating the variables in type expression T by the types A. (Generally, A will be a vector of types, one for each type variable in T .)

Type expressions are extended to denote functions from relations to relations as follows. The constant type A is read as the identity relation id_A on A. The product $R{\times}S$ of binary relations R and S of types $A \sim B$ and $C \sim D$ is defined to be the relation of type $A{\times}C \sim B{\times}D$ defined by

$$((a,c)\ ,\ (b,d))\ \in\ R{\times}S \qquad \equiv \qquad (a,b)\in R\ \ \wedge\ \ (c,d)\in S\ \ . \tag{1}$$

Finally, the function space constructor, " $\leftarrow$ ", is read as a mapping from a pair of relations R and S of types $A \sim B$ and $C \sim D$, respectively, to a binary relation $R \leftarrow S$ on functions f and g of type $A \leftarrow C$ and $B \leftarrow D$, respectively. Formally, suppose R and S are binary relations of type $A \sim B$ and $C \sim D$, respectively. Then $R \leftarrow S$ is the binary relation of type $(A \leftarrow C) \sim (B \leftarrow D)$ defined by, for all functions $f \in A \leftarrow C$ and $g \in B \leftarrow D$,

$$(f,g)\ \in\ R \leftarrow S \qquad \equiv \qquad \langle \forall\, c,d\ ::\ (f.c\ ,\ g.d) \in R\ \Leftarrow\ (c,d) \in S\rangle\ \ . \tag{2}$$

In words, f and g construct R-related values from S-related values.

As an example, suppose $(A\ ,\sqsubseteq)$ and $(B\ ,\preceq)$ are partially ordered sets. Then we can instantiate R to $\sqsubseteq$ and S to $\preceq$ getting a relation $\sqsubseteq \leftarrow \preceq$ between functions f and g of type $A \leftarrow B$. In particular, switching to the usual infix notation for membership of an ordering relation,

$$(f,f)\ \in\ \sqsubseteq \leftarrow \preceq \qquad \equiv \qquad \langle \forall\, u,v\ ::\ f.u \sqsubseteq f.v\ \Leftarrow\ u \preceq v\rangle\ \ .$$

So $(f,f) \in \sqsubseteq \leftarrow \preceq$ is the statement that f is a monotonic function. In Wadler's words, f maps $\preceq$-related values to $\sqsubseteq$-related values.

In this way, a type expression is extended to a function from relations to relations. We use $T.R$ to denote the relation obtained by instantiating the type variables in type expression T by the (vector of) relations R. Note that the construction we have given has the property that if relations R have type $A \sim B$ then $T.R$ has type $T.A \sim T.B$.

Definition 1 (Parametric Polymorphism). A term t is said to have *polymorphic* type $\langle\forall\alpha :: T.\alpha\rangle$, where T is a type expression parameterised by type variables α, if t assigns to each type A a value t_A of type $T.A$. A term t of polymorphic type $\langle\forall\alpha :: T.\alpha\rangle$ is said to be *parametrically polymorphic* if, for each instantiation of relations R to type variables, $(t_A\ ,\ t_B) \in T.R$, where R has type $A \sim B$. □

We sometimes abbreviate "parametrically polymorphic" to "parametric".

Exercise 1. The function fork creates a pair of values by simply copying its input value. That is $\mathsf{fork}.x = (x,x)$.

What is the type of fork? Formulate the property that fork is parametrically polymorphic and verify that this is indeed the case. □

Exercise 2. Function application has type $\langle\forall\, \alpha,\beta\ ::\ \alpha\ \leftarrow\ (\alpha{\leftarrow}\beta){\times}\beta\rangle$. That is, for each pair of types A and B, each function f of type $A{\leftarrow}B$ and each $b{\in}B$, $f.b \in A$. Formulate the statement that function application is parametrically polymorphic and verify that this is indeed the case. □

Exercise 3. Currying has type $\langle\forall\, \alpha,\beta,\gamma :: \alpha\ \leftarrow\ \beta\ \leftarrow\ \gamma\ \leftarrow\ (\alpha\ \leftarrow\ \beta{\times}\gamma)\rangle$. Specifically,

$$curry.f.x.y\ \ =\ \ f.(x,y)\ \ .$$

(Function application is assumed to be left-associative.) Formulate the statement that currying is parametrically polymorphic and verify that this is indeed the case. □

As we have defined them, the notions of polymorphic and parametrically polymorphic are distinct. It is common for programming languages to permit the definition of polymorphic functions without their being parametric. This is known as *ad hoc* polymorphism [13]. For example, a language may have a polymorphic binary operator that is called the "equality" operator. That is, there is a binary operator, denoted say by $==$, such that for any pair of terms $\mathtt{s}$ and $\mathtt{t}$ in the language, $\mathtt{s} == \mathtt{t}$ denotes a boolean value. However, if the operator is parametric, we deduce from its type — $Bool \leftarrow \alpha{\times}\alpha$ — that for all relations R and all $\mathtt{u}$, $\mathtt{v}$, $\mathtt{x}$ and $\mathtt{y}$,

$$(\mathtt{u} == \mathtt{v}) \;=\; (\mathtt{x} == \mathtt{y}) \qquad \Leftarrow \qquad (\mathtt{u},\mathtt{x}) \in R \;\;\wedge\;\; (\mathtt{v},\mathtt{y}) \in R \;\; .$$

In particular, taking the relation R to be an arbitrary function $\mathtt{f}$ definable in the programming language,

$$(\mathtt{f.x} \;==\; \mathtt{f.y}) \;=\; (\mathtt{x} == \mathtt{y}) \;\; .$$

In other words, if the "equality" operator is parametrically polymorphic and truly tests for equality, all functions in the language are injective — which would make the language a very restrictive language indeed. In practice, of course, "equality" operators provided in programming languages are neither parametric nor true implementations of equality on all types.

If a language is such that all polymorphic terms are parametrically polymorphic, then, by definition, if term t has type T, for each instantiation of type variables to relations R of type $A \sim B$, $(t_A\, ,\, t_B) \in T.R$. Instances of the property, for all relations R, $(t_A\, ,\, t_B) \in T.R$, are called *free theorems*. For numerous examples of free theorems, see Wadler's paper [14].

2.1 Verifiable Genericity

Polymorphism is a feature of programming languages that has many forms. Overloading and other *ad hoc* forms of polymorphism are introduced for the programmer's convenience, but offer little or no support for reasoning about programs. Parametric polymorphism, on the other hand, entails a meaningful and useful relationship between the different instances of the polymorphic object. Also, because the notion has a precise (and simple) formal definition, parametric polymorphism is a *verifiable* form of genericity.

In the remaining sections, we introduce the idea of parameterising programs by datatypes. The emphasis is on ensuring that the genericity that is introduced is clearly and precisely specified so that all claims made about its use can be formally verified.

3 Commuting Datatypes — Introduction

In this section, we introduce an example of "higher order" parametricity — parametrisation with respect to type constructors (functions from types to types, like List) rather than types (like Int). Our purpose is to demonstrate the usefulness of making this sort of abstraction, and paving the way for the theoretical development that is to follow.

We use the term "datatype" without definition for the moment. (Later it will emerge that a datatype is a "relator with membership".) Think, however, of functions from type to types like pairing, and the formation of lists or arbitrary tree structures.

The section contains a number of examples of "commuting" two datatypes. This suggests a generic property of datatypes, namely that any two datatypes can be "commuted". We argue that one should define the notion of "commuting" datatypes by a precise formulation of the properties we require of the operation (by abstraction from a suitable collection of examples). In this way, reasoning about specific programs that exploit the generic property is substantially easier.

The best known example of a commutativity property is the fact that two lists of the same length can be mapped into a single list of pairs whereby

$$([a_1\,,a_2\,,\,\ldots\,,a_n]\,,[b_1\,,b_2\,,\,\ldots\,,b_n]) \mapsto [(a_1\,,b_1)\,,(a_2\,,b_2)\,,\,\ldots\,,(a_n\,,b_n)]$$

The function that performs this operation is known as the "zip" function to functional programmers. Zip commutes a pair of lists (of the same length) into a list of pairs.

Other specific examples of commutativity properties are easy to invent. For instance, it is not difficult to imagine generalising zip to a function that commutes m lists each of length n into n lists each of length m. Indeed, this latter function is also well known under the name *matrix transposition*. Another example is the function that commutes a tree of lists all of the same length into a list of trees all of the same shape. There is also a function that "broadcasts" a value to all elements of a list —thus

$$(a\,,[b_1\,,b_2\,,\,\ldots\,,b_n]) \mapsto [(a\,,b_1)\,,(a\,,b_2)\,,\,\ldots\,,(a\,,b_n)]$$

— . That is, the datatype 'an element of type A paired with (a list of elements of type B)' is "commuted" to 'a list of (element of type A paired with an element of type B)'. More precisely, for each A, a broadcast is a parametrically polymorphic function of type $\langle\forall\alpha :: \mathsf{List}.(A{\times}\alpha) \leftarrow A \times \mathsf{List}.\alpha\rangle$; the two datatypes being "commuted" are thus ($A\times$) and List. (Note that we use the Greek letter α and the Latin letter A in order to distinguish the roles of the two parameters.)

This list broadcast is itself an instance of a subfamily of the operations that we discuss later. In general, a *broadcast* operation copies a given value to all locations in a given data structure. That is, using F to denote an arbitrary datatype and the infix operator " $\cdot$ " to denote (backward) composition of datatypes, a broadcast is a parametrically polymorphic function of type $\langle\forall\alpha :: (F\cdot(A\times)).\alpha \leftarrow ((A\times)\cdot F).\alpha\rangle$.

A final example of a generalised zip would be the (polymorphic) operation that maps values of type $(A{+}B){\times}(C{+}D)$ to values of type $(A{\times}C){+}(B{\times}D)$, i.e. commutes a product of disjoint sums to a disjoint sum of products. A necessary restriction is that the elements of the input pair of values have the same "shape", i.e. both be in the left component of the disjoint sum or both be in the right component.

In general then, a *zip* operation transforms F-structures of G-structures to G-structures of F-structures. (An F-structure is a value of type $F.X$ for some type X. So, a List-structure is just a list. Here, F could be, for example, List and G could be Tree.) Typically, "zips" are partial since they are only well-defined on structures of the same shape. As we shall see, they may also be non-deterministic; that is, a "zip" is a relation rather than a function. Finally, the arity of the two datatypes, F and G, need not be the same; for example, the classical zip function maps pairs of lists to lists of pairs, and pairing has two arguments whereas list formation has just one. (This is a complication that we will ignore here. See [6, 8] for full details.)

3.1 Structure Multiplication

A good example of the beauty of the "zip" generalisation is afforded by what we shall call "structure multiplication". (This example we owe to D.J. Lillie [private communication, December 1994].) A simple, concrete example of structure multiplication is the following. Given two lists $[a_1, a_2, \ldots]$ and $[b_1, b_2, \ldots]$ form a matrix in which the (i, j)th element is the pair (a_i, b_j). We call this "structure multiplication" because the input type is the product $\mathsf{List}.A \times \mathsf{List}.B$ for some types A and B.

Given certain basic functions, this task may be completed in one of two ways. The first way has two steps. First, the list of a's is broadcast over the list of b's to form the list

$$[([a_1, a_2, \ldots, a_n], b_1), ([a_1, a_2, \ldots, a_n], b_2), \ldots]$$

Then each b is broadcast over the list of a's. The second way is identical but for an interchange of "a" and "b".

Both methods return a list of lists, but the results are not identical. The connection between the two results is that one is the transpose of the other. The two methods and the connection between them are summarised in the following diagram.

$$\begin{tikzcd}
 & \mathsf{List}.A \times \mathsf{List}.B \arrow[dl] \arrow[dr] & \\
\mathsf{List}.(\mathsf{List}.A \times B) \arrow[d] & & \mathsf{List}.(A \times \mathsf{List}.B) \arrow[d] \\
\mathsf{List}.(\mathsf{List}.(A \times B)) \arrow[rr, leftrightarrow] & & \mathsf{List}.(\mathsf{List}.(A \times B))
\end{tikzcd}$$

The point we want to make is that there is an obvious generalisation of this procedure: replace $\mathsf{List}.A$ by $F.A$ and $\mathsf{List}.B$ by $G.B$ for some arbitrary datatypes F and G. Doing so leads to the realisation that every step involves commuting the order of a pair of datatypes.

This is made explicit in the diagram below where each arrow is labelled by two datatypes separated by the symbol "$\leftrightarrow$". For example, the middle, bottom arrow is labelled "$F \leftrightarrow G$". This indicates a transformation that "commutes" an "F-structure" of "G-structures" into a G-structure of F-structures. The rightmost arrow is labelled "$A\times \leftrightarrow G$". An ($A\times$)-structure is a pair of elements of which the first is of type A. Ignoring the "$F.$" for the moment, the transformation requires values of type $A \times G.B$ to be transformed into values of type $G.(A \times B)$. That is, an ($A\times$)-structure of G-structures has to be transformed into a G-structure of ($A\times$)-structures. Taking the "$F.$" into account simply means that the transformation has to take place at each place in the F-structure.

$$\begin{tikzcd}
& F.A\times G.B \arrow[dl, "((F.A)\times)\leftrightarrow G"'] \arrow[dr, "(\times(G.B))\leftrightarrow F"] & \\
G.(F.A\times B) \arrow[d, "\times B\leftrightarrow F"'] & & F.(A\times G.B) \arrow[d, "A\times\leftrightarrow G"] \arrow[dll, "F\cdot(A\times)\leftrightarrow G" description] \\
G.(F.(A\times B)) & & F.(G.(A\times B)) \arrow[ll, "F\leftrightarrow G"]
\end{tikzcd}$$

An additional edge has been added to the diagram to show the usefulness of generalising the notion of commutativity beyond just broadcasting; this additional inner edge shows how the commutativity of the diagram can be decomposed into smaller parts[2].

Filling out the requirement in more detail, let us suppose that, for datatypes F and G, $\mathsf{zip}.F.G$ is a family of relations indexed by types such that $(\mathsf{zip}.F.G)_A$ has type $(G{\cdot}F).A \sim (F{\cdot}G).A$. Then, assuming that, whenever $R : U \sim V$, $F.R : F.U \sim F.V$, the transformations that are used are as shown in the diagram below.

[2] The additional edge together with the removal of the right-pointing edge in the bottom line seem to make the diagram asymmetric. But, of course, there are symmetric edges. Corresponding to the added diagonal edge there is an edge connecting $G.(F.A \times B)$ and $F.(G.(A \times B))$ but only one of these edges is needed in the argument that follows.

$$\begin{tikzcd}
& F.A \times G.B \arrow[dl, "(\mathsf{zip}.((F.A)\times).G)_B"'] \arrow[dr, "(\mathsf{zip}.(\times(G.B)).F)_A"] & \\
G.(F.A \times B) \arrow[d, "G.(\mathsf{zip}.(\times B).F)_A"'] & & F.(A \times G.B) \arrow[dll, "(\mathsf{zip}.(F\cdot(A\times)).G)_B"'] \arrow[d, "F.(\mathsf{zip}.(A\times).G)_B"] \\
G.(F.(A \times B)) & & F.(G.(A \times B)) \arrow[ll, "(\mathsf{zip}.F.G)_{A\times B}"]
\end{tikzcd}$$

Now, in order to show that the whole diagram commutes (in the standard categorical sense of commuting diagram) it suffices to show that the two smaller diagrams commute. Specifically, the following two equalities must be established:

$$(\mathsf{zip}.(F\cdot(A\times)).G)_B \;=\; (\mathsf{zip}.F.G)_{A\times B} \circ F.(\mathsf{zip}.(A\times).G)_B \tag{3}$$

and

$$\begin{aligned} & (\mathsf{zip}.(F\cdot(A\times)).G)_B \circ (\mathsf{zip}.(\times(G.B)).F)_A \\ = \;\; & G.(\mathsf{zip}.(\times B).F)_A \circ (\mathsf{zip}.((F.A)\times).G)_B \;\; . \end{aligned} \tag{4}$$

We shall in fact design our definition of "commuting datatypes" in such a way that these two equations are satisfied (almost) by definition. In other words, our notion of "commuting datatypes" is such that the commutativity of the above diagram is automatically guaranteed.

3.2 Broadcasts

A family of functions bcst, where $\mathsf{bcst}_{A,B} \;:\; F.(A\times B) \leftarrow F.A\times B$, is said to be a *broadcast* for datatype F iff it is parametrically polymorphic in the parameters A and B and $\mathsf{bcst}_{A,B}$ behaves coherently with respect to product in the following sense. First, the diagram

$$\begin{tikzcd}
F.(A \times \mathbb{1}) \arrow[dr, "(F\cdot\mathsf{rid})_A"'] & & F.A \times \mathbb{1} \arrow[ll, "\mathsf{bcst}_{A,\mathbb{1}}"'] \arrow[dl, "(\mathsf{rid}\cdot F)_A"] \\
& F.A &
\end{tikzcd}$$

(where $\mathsf{rid}_A : A \leftarrow A\times \mathbb{1}$ is the obvious natural isomorphism) commutes. Second, the diagram

$$\begin{tikzcd}
F.A\times(B\times C) \arrow[dd, "\mathsf{bcst}_{A\,,\,B\times C}"'] & (F.A\times B)\times C \arrow[l, "\mathsf{ass}_{F.A,B,C}"'] \arrow[d, "\mathsf{bcst}_{A,B}\times \mathsf{id}_C"] \\
 & F.(A\times B)\times C \arrow[d, "\mathsf{bcst}_{A\times B\,,\,C}"] \\
F.(A\times(B\times C)) & F.((A\times B)\times C) \arrow[l, "F{\cdot}\mathsf{ass}_{A,B,C}"]
\end{tikzcd}$$

(where $\mathsf{ass}_{A,B,C} : A\times(B\times C) \leftarrow (A\times B)\times C$ is the obvious natural isomorphism) commutes as well.

The idea behind a "broadcast" is very simple. A broadcast for datatype F is a way of duplicating a given value of type B across every location in an F-structure of A's. The broadcasting operation is what Moggi [11] calls the "strength" of a datatype.

The type of $\mathsf{bcst}_{A,B}$ of datatype F is the same as the type of $(\mathsf{zip}.(\times B).F)_A$, namely $F.(A\times B) \sim F.A\times B$. We give a generic specification of a zip such that, if F and the family of datatypes $(\times A)$ are included in a class of commuting datatypes, then any relation satisfying the requirements of $(\mathsf{zip}.(\times B).F)_A$ also satisfies the definition of $\mathsf{bcst}_{A,B}$ (and is a total function).

Let us begin with an informal scrutiny of the definition of broadcast. In the introduction to this section we remarked that a broadcast operation is an example of a zip. Specifically, a broadcast operation is a zip of the form $(\mathsf{zip}.(\times A).F)_B$. That is, a broadcast commutes the datatypes $(\times A)$ and F. Moreover, paying due attention to the fact that the datatype F is a parameter of the definition, we observe that all the transformations involved in the definition of a broadcast are themselves special cases of a broadcast operation and thus of zips.

In the first diagram there are two occurrences of the canonical isomorphism rid. In general, we recognise a projection of type $A \leftarrow A\times B$ as a broadcast where the parameter F is instantiated to K_A, the datatype that is constantly A when applied to types. Thus rid_A is $(\mathsf{zip}.(\times\mathbb{1}).K_A)_B$ for some arbitrary B. In words, rid_A commutes the datatypes $(\times\mathbb{1})$ and K_A. Redrawing the first diagram above, using that all the arrows are broadcasts and thus zips, we get the following diagram[3].

[3] To be perfectly correct we should instantiate each of the transformations at some arbitrary B. We haven't done so because the choice of which B in this case is truly irrelevant.

$$\begin{tikzcd}
F.(A\times \mathbb{1}) \arrow[dr, "F.(\mathsf{zip}.(\times\mathbb{1}).(K_A))"'] & & F.A\times\mathbb{1} \arrow[ll, "(\mathsf{zip}.(\times\mathbb{1}).F)\cdot K_A"'] \arrow[dl, "\mathsf{zip}.(\times\mathbb{1}).(K_{F.A})"] \\
& F.A &
\end{tikzcd}$$

Expressed as an equation, this is the requirement that

$$\mathsf{zip}.(\times\mathbb{1}).(K_{F.A}) \;=\; F.(\mathsf{zip}.(\times\mathbb{1}).(K_A)) \circ ((\mathsf{zip}.(\times\mathbb{1}).F)\cdot K_A) \qquad (5)$$

Now we turn to the second diagram in the definition of boradcasts. Just as we observed that rid is an instance of a broadcast and thus a zip, we also observe that ass is a broadcast and thus a zip. Specifically, $\mathsf{ass}_{A,B,C}$ is $(\mathsf{zip}.(\times C).(A\times))_B$. Once again, every edge in the diagram involves a zip operation! That is not all. Yet more zips can be added to the diagram. For our purposes it is crucial to observe that the bottom left and middle right nodes —the nodes labelled $F.(A\times(B\times C))$ and $F.(A\times B)\times C$— are connected by the zip operation $(\mathsf{zip}.(\times C).(F.(A\times)))_B$.

$$\begin{tikzcd}
F.A\times(B\times C) \arrow[dd, "(\mathsf{zip}.(\times(B\times C)).F)_A"'] & (F.A\times B)\times C \arrow[l, "(\mathsf{zip}.(\times C).((F.A)\times))_B"'] \arrow[d, "(\mathsf{zip}.(\times B).F)_A\times\mathsf{id}_C"] \\
& F.(A\times B)\times C \arrow[dl, "(\mathsf{zip}.(\times C).(F\cdot(A\times)))_B" description] \arrow[d, "(\mathsf{zip}.(\times C).F)_{A\times B}"] \\
F.(A\times(B\times C)) & F.((A\times B)\times C) \arrow[l, "F.(\mathsf{zip}.(\times C).(A\times))_B"]
\end{tikzcd}$$

This means that we can decompose the original coherence property into a combination of two properties of zips. These are as follows. First, the lower triangle:

$$(\mathsf{zip}.(\times C).(F\cdot(A\times)))_B \;=\; F.(\mathsf{zip}.(\times C).(A\times))_B \circ (\mathsf{zip}.(\times C).F)_{A\times B} \quad . \qquad (6)$$

Second, the upper rectangle:

$$\begin{aligned}
& (\mathsf{zip}.(\times(B\times C)).F)_A\circ(\mathsf{zip}.(\times C).((F.A)\times))_B \\
= \;& (\mathsf{zip}.(\times C).(F\cdot(A\times)))_B\circ(\mathsf{zip}.(\times B).F)_A\times\mathsf{id}_C \ . \qquad (7)
\end{aligned}$$

Note the strong similarity between (5) and (6). They are both instances of one equation parameterised by three different datatypes. There is also a similarity between these two equations and (3); the latter is an instance of the same parameterised equation after taking the converse of both sides and assuming that $\mathsf{zip}.F.G = (\mathsf{zip}.G.F)^{\cup}$. Less easy to spot is the similarity between (4) and (7). As we shall see, however, both are instances of one equation parameterised again by three different datatypes except that (7) is obtained by applying the converse operator to both sides of the equation and again assuming that $\mathsf{zip}.F.G = (\mathsf{zip}.G.F)^{\cup}$.

4 Allegories and Relators

In this section, we begin the development of theory of datatypes in which the primitive mechanisms for defining programs are all parametrically polymorphic.

We use relation algebra (formally "allegory" theory [5]) as the basis for our theory. There are many reasons why we choose relations as basis rather than functions. The most compelling reason is that, for us, programming is about constructing implementations of input-output relations. Program specifications are thus relations; but, also, programs themselves may be relations rather than functions — if one admits non-determinism, which we certainly wish to do.

Of more immediate relevance to generic programming is that, as we have seen, the formal definition of parametric polymorphism necessitates an extension of the definition of type constructors allowing them to map relations to relations. (The extension to relations is unavoidable because, even for functions f and g , $f \leftarrow g$ is a *relation* on functions and not a function from functions to functions. Also, the definition of commuting datatypes necessarily involves both nondeterminism and partiality — in particular, when one of the datatypes is the constant datatype K_A , for some type A .

In this section, we build up a language of terms defining relational specifications and implementations. The primitive relations in this language are not considered; our concern is with defining a number of constructs that build composite relations from given primitive relations and that preserve functionality and are parametrically polymorphic.

4.1 Allegories

An *allegory* is a category with additional structure, the additional structure capturing the most essential characteristics of relations. Being a category means, of course, that for every object A there is an identity arrow id_A , and that every pair of arrows $R : A \leftarrow B$ and $S : B \leftarrow C$, with matching source and target[4], can be composed: $R \circ S : A \leftarrow C$. Composition is associative and has id as a unit.

The additional axioms are as follows. First of all, arrows of the same type are ordered by the *partial order* $\subseteq$ and composition is monotonic with respect to this order. That is,

$$S_1 \circ T_1 \subseteq S_2 \circ T_2 \quad \Leftarrow \quad S_1 \subseteq S_2 \ \wedge \ T_1 \subseteq T_2 \quad .$$

[4] Note that we refer to the "source" and "target" of an arrow in a category in order to avoid confusion with the domain and range of a relation introduced later.

Secondly, for every pair of arrows $R, S : A \leftarrow B$, their *intersection* (*meet*) $R \cap S$ exists and is defined by the following universal property, for each $X : A \leftarrow B$,

$$X \subseteq R \wedge X \subseteq S \quad \equiv \quad X \subseteq R \cap S \quad .$$

Finally, for each arrow $R : A \leftarrow B$ its *converse* $R^{\cup} : B \leftarrow A$ exists. The converse operator is defined by the requirements that it is its own Galois adjoint, that is,

$$R^{\cup} \subseteq S \quad \equiv \quad R \subseteq S^{\cup} \quad ,$$

and is contravariant with respect to composition,

$$(R \circ S)^{\cup} = S^{\cup} \circ R^{\cup} \quad .$$

All three operators of an allegory are connected by the *modular law*, also known as Dedekind's law [12]:

$$R \circ S \cap T \subseteq (R \cap T \circ S^{\cup}) \circ S \quad .$$

The standard example of an allegory is Rel, the allegory with sets as objects and relations as arrows. With this allegory in mind, we refer henceforth to the arrows of an allegory as “relations”.

(Note that we no longer use the symbol “ $\sim$ ” in addition to the symbol “ $\leftarrow$ ” as a means of distingushing relations from functions. From here on, relations are our prime interest, unless we specifically say otherwise.)

4.2 Relators

Now that we have the definition of an allegory we can give the definition of a relator.

Definition 2 (Relator). A *relator* is a monotonic functor that commutes with converse. That is, let $\mathcal{A}$ and $\mathcal{B}$ be allegories. Then the mapping $F : \mathcal{A} \leftarrow \mathcal{B}$ is a relator iff,

$$F.R : F.A \leftarrow F.B \quad \Leftarrow \quad R : A \leftarrow B \quad , \tag{8}$$

$$F.R \circ F.S = F.(R \circ S) \quad \text{for each } R : A \leftarrow B \text{ and } S : B \leftarrow C \quad , \tag{9}$$

$$F.\mathsf{id}_A = \mathsf{id}_{F.A} \quad \text{for each object } A \quad , \tag{10}$$

$$F.R \subseteq F.S \quad \Leftarrow \quad R \subseteq S \quad \text{for each } R : A \leftarrow B \text{ and } S : A \leftarrow B \quad , \tag{11}$$

$$(F.R)^{\cup} = F.(R^{\cup}) \quad \text{for each } R : A \leftarrow B \quad . \tag{12}$$

□

Two examples of relators have already been given. List is a unary relator, and product is a binary relator. List is an example of an inductively-defined datatype; in [1] it was observed that all inductively-defined datatypes are relators.

A design requirement which led Backhouse to the above definition of a relator [1, 2] is that a relator should extend the notion of a functor but in such a way that it coincides with the latter notion when restricted to functions. Formally, relation $R : A \leftarrow B$ is *total* iff

$$\mathsf{id}_B \subseteq R^\cup \circ R \quad ,$$

and relation R is single-valued or *simple* iff

$$R \circ R^\cup \subseteq \mathsf{id}_A \quad .$$

A *function* is a relation that is both total and simple. It is easy to verify that total and simple relations are closed under composition. Hence, functions are closed under composition too. In other words, the functions form a sub-category. For an allegory $\mathcal{A}$, we denote the sub-category of functions by $Map(\mathcal{A})$. In particular, $Map(\mathsf{Rel})$ is the category having sets as objects and functions as arrows. Now the desired property of relators is that relator $F : \mathcal{A} \leftarrow \mathcal{B}$ is a functor of type $Map(\mathcal{A}) \leftarrow Map(\mathcal{B})$. It is easily shown that our definition of relator guarantees this property. For instance, that relators preserve totality is proved as follows:

$$\begin{array}{cl}
 & (F.R)^\cup \circ F.R \\
= & \quad \{ \quad \text{converse and relators commute: (12)} \quad \} \\
 & F.(R^\cup) \circ F.R \\
= & \quad \{ \quad \text{relators distribute through composition: (9)} \quad \} \\
 & F.(R^\cup \circ R) \\
\supseteq & \quad \{ \quad \text{assume } \mathsf{id}_B \subseteq R^\cup \circ R, \\
 & \qquad\quad \text{relators are monotonic: (11)} \quad \} \\
 & F.\mathsf{id}_B \\
= & \quad \{ \quad \text{relators preserve identities: (10)} \quad \} \\
 & \mathsf{id}_{F.B} \quad .
\end{array}$$

The proof that relators preserve simplicity is similar.

Note how this little calculation uses all four requirements for F to be a relator.

The following characterisation of functions proves to be very useful (for one reason because it is a Galois connection).

Definition 3 (Function). An arrow $f : A \leftarrow B$ is a *function* if, for all arrows, $R : C \leftarrow B$ and $S : A \leftarrow C$,

$$R \circ f^\cup \subseteq S \;\equiv\; R \subseteq S \circ f \quad .$$

□

Exercise 4. Show that definition 3 is equivalent to the conjunction of the two properties $f \circ f^\cup \subseteq \mathsf{id}_A$ and $\mathsf{id}_B \subseteq f^\cup \circ f$. □

4.3 Composition and Relators Are Parametric

As we go along, we introduce various building blocks for constructing specifications and programs. As these are introduced, we check whether they are parametric. Composition and relators are two such building blocks. So, in this section, we formulate and verify their parametricity properties.

This is, in fact, very straightforward. The advantage of using point-free relation algebra is that we can give a succinct definition of the arrow operator, leading to more succinct formulations of its basic properties and, hence, easier proofs.

There are several equivalent ways of defining the arrow operator, all of which are connected by definition 3. The one obtained directly from the pointwise definition is:

$$(f,g) \in R{\leftarrow}S \;\equiv\; f^{\cup} \circ R \circ g \supseteq S \;. \tag{13}$$

Another form, which we sometimes use, is:

$$(f,g) \in R{\leftarrow}S \;\equiv\; R{\circ}g \supseteq f{\circ}S \;. \tag{14}$$

Now, a relator F has type $\langle\forall\,\alpha,\beta :: (F.\alpha \leftarrow F.\beta) \leftarrow (\alpha \leftarrow \beta)\rangle$. (The two occurrences "F" in "$F.\alpha$" and "$F.\beta$" are maps from objects to objects, the first occurrence is a map from arrows to arrows.) Parametricity thus means that., for all relations R and S , and all functions f and g ,

$$(F.f\,,\,F.g) \in F.R \leftarrow F.S \quad\Leftarrow\quad (f,g) \in R{\leftarrow}S \;.$$

Here we have used the pointwise definition (2) of the arrow operator. Now using the point-free definition (13), this reduces to:

$$(F.f)^{\cup} \circ F.R \circ F.g \supseteq F.S \quad\Leftarrow\quad f^{\cup} \circ R \circ g \supseteq S \;.$$

This we verify as follows:

$$\begin{array}{ll}
 & (F.f)^{\cup} \circ F.R \circ F.g \\
= & \quad\{ \quad \text{converse commutes with relators: (12)} \quad \} \\
 & F.(f^{\cup}) \circ F.R \circ F.g \\
= & \quad\{ \quad \text{relators distribute through composition: (9)} \quad \} \\
 & F.(f^{\cup} \circ R \circ g) \\
\supseteq & \quad\{ \quad \text{assume } f^{\cup} \circ R \circ g \supseteq S \\
 & \quad\phantom{\{} \quad \text{relators are monotonic: (11)} \quad \} \\
 & F.S \;.
\end{array}$$

Note how this little calculation uses three of the requirements of F being a relator. So, by design, relators preserve functionality and are parametric.

Exercise 5. Formulate and verify the property that composition is parametrically polymorphic. □

4.4 Division and Tabulation

The allegory Rel has more structure than we have captured so far with our axioms. For instance, in Rel we can take arbitrary unions (joins) of relations. There are also two "division" operators, and Rel is "tabulated". In full, Rel is a unitary, tabulated, locally complete, division allegory. For full discussion of these concepts see [5] or [4]. Here we briefly summarise the relevant definitions.

We say that an allegory is *locally complete* if for each set $\mathcal{S}$ of relations of type $A \leftarrow B$, the union $\cup\mathcal{S} : A \leftarrow B$ exists and, furthermore, intersection and composition distribute over arbitrary unions. The defining property of union is that, for all $X : A \leftarrow B$,

$$\cup\mathcal{S} \subseteq X \equiv \forall(S \in \mathcal{S} :: \; S \subseteq X) \quad .$$

We use the notation $\perp\!\!\perp_{A,B}$ for the smallest relation of type $A \leftarrow B$ and $\top\!\!\top_{A,B}$ for the largest relation of the same type.

The existence of a largest relation for each pair of objects A and B is guaranteed by the existence of a "unit" object, denoted by $\mathbb{1}$. We say that object $\mathbb{1}$ is a *unit* if $\mathsf{id}_{\mathbb{1}}$ is the largest relation of its type and for every object A there exists a total relation $!_A : \mathbb{1} \leftarrow A$. If an allegory has a unit then it is said to be *unitary*.

The most crucial consequence of the distributivity of composition over union is the existence of two so-called *division* operators "$\backslash$" and "$/$". Specifically, we have the following three Galois-connections. For all $R : A \leftarrow B$, $S : B \leftarrow C$ and $T : A \leftarrow C$,

$$\begin{aligned} R{\circ}S \subseteq T &\quad\equiv\quad S \subseteq R\backslash T \quad , \\ R{\circ}S \subseteq T &\quad\equiv\quad R \subseteq T/S \quad , \\ S \subseteq R\backslash T &\quad\equiv\quad R \subseteq T/S \quad , \end{aligned}$$

(where, of course, the third is just a combination of the first two).

Note that $R\backslash T : B \leftarrow C$ and $T/S : A \leftarrow B$. The interpretation of the factors is

$$\begin{aligned} (b{,}c) \in R\backslash T &\quad\equiv\quad \forall(a : \; (a{,}b) \in R : \; (a{,}c) \in T) \quad , \\ (a{,}b) \in T/S &\quad\equiv\quad \forall(c : \; (b{,}c) \in S : \; (a{,}c) \in T) \quad . \end{aligned}$$

The final characteristic of Rel is that it is "tabular". That is, each relation is a set of ordered pairs. Formally, we say that an object C and a pair of functions $f : A \leftarrow C$ and $g : B \leftarrow C$ is a *tabulation* of relation $R : A \leftarrow B$ if

$$R = f{\circ}g^{\cup} \quad\wedge\quad f^{\cup}{\circ}f \cap g^{\cup}{\circ}g \;=\; \mathsf{id}_C \quad .$$

An allegory is said to be *tabular* if every relation has a tabulation.

Allegory Rel is tabular. Given relation $R : A \leftarrow B$, define C to be the subset of the cartesian product $A{\times}B$ containing the pairs of elements for which $(x{,}y) \in R$. Then the pair of projection functions $\mathsf{outl} : A \leftarrow C$ and $\mathsf{outr} : B \leftarrow C$ is a tabulation of R.

If allegory $\mathcal{B}$ is tabular, a functor commutes with converse if it is monotonic. (Bird and De Moor [4] claim an equivalence but their proof is flawed.) So, if we define a relator on a tabular allegory, one has to prove just requirement (11), from which (12) can be deduced. Property (12) is, however, central to many calculations and can usually be easily established without an appeal to tabularity. The same holds for property (9) which —in a tabulated allegory— is a consequence of (12) and the fact that relators are functors in the underlying category of maps: the property is also central to many calculations and can usually be easily established without an appeal to tabularity. For this reason we prefer to retain the definition that we originally proposed.

4.5 Domains

In addition to the source and target of a relation it is useful to know their domain and range. The *domain* of a relation $R : A \leftarrow B$ is that subset $R\ \ $ of id_B defined by the Galois connection:

$$R \subseteq \top\!\!\top_{A,B} \circ X \ \equiv\ R\ \ \subseteq X \quad \text{for each } X \subseteq \mathsf{id}_B. \tag{15}$$

The *range* of $R : A \leftarrow B$, which we denote by $R\ \ $, is the domain of $R^{\cup}$.

The interpretation of the domain of a relation is the set of all y such that $(x,y) \in R$ for some x. We use the names "domain" and "range" because we usually interpret relations as transforming "input" y on the right to "output" x on the left. The domain and range operators play an important role in a relational theory of datatypes.

5 Datatype = Relator + Membership

This section defines a class of relators that we call the "regular" relators. This class contains a number of primitive relators —the constant relators, one for each type, product and coproduct— and a number of composite relators. We also introduce the notion of the "membership" relation. The regular relators correspond to the (non-nested) datatypes that one can define in a functional programming language. They all have an associated membership relation; this leads to the proposal, first made by De Moor and Hoogendijk [7], that a datatype is a relator with membership.

There are two basic means for forming composite relators, namely induction and pointwise closure. We begin with a brief discussion of pointwise closure, since we need to say something about it, but do not go into it here.

5.1 Pointwise Closure

For the purposes of this discussion, let us take for granted that List is an endorelator and product (denoted by an infix " $\times$ ") is a binary relator. By "endo", we mean that the source and target allegories of List are the same. By "binary", we mean that, for some allegory $\mathcal{A}$, product maps pairs of objects of $\mathcal{A}$ to an object of $\mathcal{A}$, and pairs of arrows of $\mathcal{A}$ to an arrow of $\mathcal{A}$.

From these two relators, we can form new relators by so-called "pointwise closure". One simple way is composition: defining $\mathsf{List}{\cdot}\mathsf{List}$ by, for all X (whether an arrow or an object), $(\mathsf{List}{\cdot}\mathsf{List}).X = \mathsf{List}.(\mathsf{List}.X)$, it is easily checked that $\mathsf{List}{\cdot}\mathsf{List}$ is a relator. Indeed, it is easy to verify that the functional composition of two relators $F : \mathcal{A} \leftarrow \mathcal{B}$ and $G : \mathcal{B} \leftarrow \mathcal{C}$, which we denote by $F{\cdot}G$, is a relator. There is also an identity relator for each allegory $\mathcal{A}$, which we denote by Id leaving the specific allegory to be inferred from the context. The relators thus form a category —a fact that we need to bear in mind later— .

Another way to form new relators is by tupling and projection. Tupling permits the definition of relators that are multiple-valued. So far, all our examples of relators have been single-valued. Modern functional programming languages provide a syntax whereby relators (or, more precisely, the corresponding functors) can de defined as datatypes. Often datatypes are single-valued, but in general they are not. Mutually-recursive datatypes are commonly occurring, programmer-defined datatypes that are not single-valued. But composite-valued relators also occur in the definition of single-valued relators. For example, the (single-valued) relator F defined by $F.R = R{\times}R$ is the composition of the relator $\times$ after the (double-valued) doubling relator. More complicated examples like the binary relator $\otimes$ that maps the pair (R,S) to $R{\times}S{\times}S$ involve projection (of the pair (R,S) onto components R and S) as well as repetition (doubling), and product. The programmer is not usually aware of this because the use of multiple-valued relators is camouflaged by the use of variables. For our purposes, however, we need a variable-free mechanism for composing relators. This is achieved by making the arity of a relator explicit and introducing mechanisms for tupling and projection.

In order to make all this precise, we need to consider a collection of allegories created by closing some base allegory $\mathcal{C}$ under the formation of finite cartesian products. (The cartesian product of two allegories, defined in the usual pointwise fashion, is clearly an allegory. Moreover, properties such as unitary, locally complete etc. are preserved in the process.) An allegory in the collection is thus $\mathcal{C}^k$ where k, the *arity* of the allegory is either a natural number or $l{*}m$ where l is an arity and m is a number.

The *arity* of a relator F is $k \leftarrow l$ if the target of F is $\mathcal{C}^k$ and its source is $\mathcal{C}^l$. We write $F : k \leftarrow l$ rather than the strictly correct $F : \mathcal{C}^k \leftarrow \mathcal{C}^l$. A relator with arity $1 \leftarrow 1$ is called an *endorelator* and a relator with arity $1 \leftarrow k$, for some k, is called *single-valued*.

To our collection of primitive relators, we need to add "duplicating" relators, Δ_k, of arity $k{*}m \leftarrow k$ (that simply duplicate their arguments m times) and projection relators, $Proj_m$, of arity $k \leftarrow k{*}m$ (that project vectors onto their components).

By using variables, tupling and projection are made invisible; functions defined on the primitive components are implicitly extended "pointwise" to composites formed by these relators.

Although the process of pointwise closure is seemingly straightforward, it is a non-trivial exercise to formulate it precisely, and the inclusion of arity information complicates the discussion substantially. For this reason, we restrict attention almost exculsively to endorelators. Occasionally, we illustrate how tupling is handled using the case of a pair of relators. For this case, if F and G are both endorelators , we use $F\Delta G$ to denote the relator of arity $2\leftarrow 1$ such that $(F\Delta G).R = (F.R\,,\,G.R)$. Complete accounts can be found in [8, 6].

Definition 4 (Pointwise Closed). A collection of relators is said to be *pointwise closed* with *base allegory* $\mathcal{C}$ if each relator in the collection has type $\mathcal{C}^k\leftarrow\mathcal{C}^l$ for some arities k and l , and the collection includes all projections, and is closed under functional composition and tupling. □

5.2 Regular Relators

The "regular relators" are those relators constructed from three primitive (classes of) relators by pointwise closure and induction.

For each object A in an allegory, there is a relator K_A defined by $K_A.R = \mathsf{id}_A$. Such relators are called *constant* relators.

A *coproduct* of two objects consists of an object and two injection relations. The object is denoted by $A{+}B$ and the two relations by $\mathsf{inl}_{A,B} \,:\, A{+}B\leftarrow A$ and $\mathsf{inr}_{A,B} \,:\, A{+}B\leftarrow B$. For the injection relations we require that

$$\mathsf{inl}^{\cup}_{A,B}\circ\mathsf{inl}_{A,B} = \mathsf{id}_A \quad\text{and}\quad \mathsf{inr}^{\cup}_{A,B}\circ\mathsf{inr}_{A,B} = \mathsf{id}_B \quad , \tag{16}$$

$$\mathsf{inl}^{\cup}_{A,B}\circ\mathsf{inr}_{A,B} = \perp\!\!\!\perp_{A,B} \quad , \tag{17}$$

and

$$(\mathsf{inl}_{A,B}\circ\mathsf{inl}^{\cup}_{A,B}) \cup (\mathsf{inr}_{A,B}\circ\mathsf{inr}^{\cup}_{A,B}) \;=\; \mathsf{id}_{A+B} \quad . \tag{18}$$

Having the functions inl and inr, we can define the *junc* operator: for all $R : C\leftarrow A$ and $S : C\leftarrow B$,

$$R\triangledown S \;=\; (R\circ\mathsf{inl}^{\cup}_{A,B}) \cup (S\circ\mathsf{inr}^{\cup}_{A,B}) \quad , \tag{19}$$

and the *coproduct relator*: for all $R : C\leftarrow A$ and $S : D\leftarrow B$

$$R{+}S \;=\; (\mathsf{inl}_{C,D}\circ R)\triangledown(\mathsf{inr}_{C,D}\circ S) \quad .$$

In the future, we adopt the convention that " $\triangledown$ " and " $+$ " have equal precedence, which is higher than the precedence of composition, which, in turn, is higher than the precedence of set union. We will also often omit type information (the subscripts attached to constants like inl).

Exercise 6. Prove the following properties of the junc operator:

$$R\triangledown S\subseteq X \;\equiv\; R\subseteq X\circ\mathsf{inl} \wedge S\subseteq X\circ\mathsf{inr} \quad ,$$

the *computation* rules

$$R{\triangledown}S \circ \mathsf{inl} = R \quad \text{, and}$$

$$R{\triangledown}S \circ \mathsf{inr} = S \quad ,$$

and

$$X \subseteq R{\triangledown}S \;\equiv\; X{\circ}\mathsf{inl} \subseteq R \wedge X{\circ}\mathsf{inr} \subseteq S \quad .$$

Hence conclude the *universal property*

$$X = R{\triangledown}S \;\equiv\; X{\circ}\mathsf{inl} = R \wedge X{\circ}\mathsf{inr} = S \quad .$$

Prove, in addition, the *fusion* laws:

$$Q \circ R{\triangledown}S = (Q{\circ}R){\triangledown}(Q{\circ}S) \quad ,$$

$$R{\triangledown}S \circ T{+}U = (R{\circ}T){\triangledown}(S{\circ}U) \quad .$$

Formulate and verify the parametricity property of " $\triangledown$ " . Show also that " $+$ " is a relator. □

A *product* of two objects consists of an object and two projection arrows. The object is denoted by $A{\times}B$ and the two arrows by $\mathsf{outl}_{A,B} : A \leftarrow A{\times}B$ and $\mathsf{outr}_{A,B} : B \leftarrow A{\times}B$. For the arrows we require them to be functions and that

$$\mathsf{outl}_{A,B} \circ \mathsf{outr}^{\cup}_{A,B} = \top\!\!\top_{A,B} \quad , \tag{20}$$

and

$$\mathsf{outl}^{\cup}_{A,B} \circ \mathsf{outl}_{A,B} \;\cap\; \mathsf{outr}^{\cup}_{A,B} \circ \mathsf{outr}_{A,B} \;=\; \mathsf{id}_{A\times B} \quad . \tag{21}$$

Properties (20) and (21) are the same as saying that $A{\times}B$, $\mathsf{outl}_{A,B}$ and $\mathsf{outr}_{A,B}$, together, tabulate $\top\!\!\top_{A,B}$. Recalling that a unitary allegory is an allegory in which $\top\!\!\top_{A,B}$ exists for each pair of objects, A , B , it follows that products exist in a unitary, tabular allegory. (Note, however, that it is not necessary for all arrows to have a tabulation for products to exist.)

Having the projection functions outl and outr , we can define the *split* operator[5] on relations: for all $R : A \leftarrow C$ and $S : B \leftarrow C$

$$R{\vartriangle}S \;=\; \mathsf{outl}^{\cup}_{A,B} \circ R \;\cap\; \mathsf{outr}^{\cup}_{A,B} \circ S \quad , \tag{22}$$

and the *product relator*: for all for $R : C \leftarrow A$ and $S : D \leftarrow B$,

$$R{\times}S \;=\; (R \circ \mathsf{outl}_{A,B}) \vartriangle (S \circ \mathsf{outr}_{A,B}) \quad .$$

Exercise 7. In category theory, coproduct and product are duals. Our definitions of product and coproduct are dual in the sense that compositions are reversed (as in category theory), set union is replaced by set intersection, and the

[5] The symbol " Δ " is conventionally used in category theory to denote the doubling functor. Its similarity with the symbol " $\vartriangle$ ", which we use to denote the split operator, is by design and not coincidence. Category theoreticians, however, use $\langle R,S \rangle$ instead of $R{\vartriangle}S$, thereby failing to highlight the correspondence. They also use $[R,S]$ where we use $R{\triangledown}S$, thus obscuring the duality.

empty relation, $\bot\!\!\bot$, is replaced by the universal relation, $\top\!\!\top$. The properties of the two are not completely dual, however, because composition distributes through set union but does not distribute through set intersection. This makes proofs about product and split more difficult than proofs about coproduct and junc. Nevertheless, crucial properties do dualise. In particular, product is a relator, and split is parametric.

This exercise is about proving these properties omitting the harder parts. For the full proofs see [1, 2, 8].

[Hard] Prove the *computation* rules

$$\mathsf{outl} \circ R \triangle S = R \circ S \quad \text{, and}$$

$$\mathsf{outr} \circ R \triangle S = S \circ R \quad .$$

(Hint: use the modular identity and the fact that outl is a function. You should also use the identity $R \circ S \;\; = R \cap \top\!\!\top \circ S$.)

Assuming the *split-cosplit rule*

$$(R \triangle S)^{\cup} \circ T \triangle U = (R^{\cup} \circ T) \cap (S^{\cup} \circ U) \quad ,$$

prove the fusion laws

$$R \times S \circ T \triangle U = (R \circ T) \triangle (S \circ U) \quad \text{, and}$$

$$R \times S \circ T \times U = (R \circ T) \times (S \circ U) \quad .$$

Fill in the remaining elements of the proof that " $\times$ " is a relator. Finally, formulate and verify the parametricity property of " $\triangle$ " . □

Tree relators are defined as follows. Suppose that relation $\mathsf{in} : A \leftarrow F.A$ is an initial F-algebra. That is to say, suppose that for each relation $R : B \leftarrow F.B$ (thus each " F-algebra") there exists a unique $F-$ homomorphism to R from in . We denote this unique homomorphism by $(\!|F\,;R|\!)$ and call it a *catamorphism.* Formally, $(\!|F\,;R|\!)$ and in are characterized by the universal property that, for each relation $X : B \leftarrow A$ and each relation $R : B \leftarrow F.B$,

$$X = (\!|F\,;R|\!) \quad \equiv \quad X \circ \mathsf{in} = R \circ F.X \quad . \tag{23}$$

Now, let $\otimes$ be a binary relator and assume that, for each A ,

$$\mathsf{in}_A \;\; : \;\; T.A \leftarrow A \otimes T.A$$

is an initial algebra of ($A\otimes$)[6]. Then the mapping T defined by, for all $R : A \leftarrow B$,

$$T.R = (\!|A\otimes\,;\; \mathsf{in}_B \circ R \otimes \mathsf{id}_{T.B}|\!)$$

is a relator, *the tree relator induced by* $\otimes$.

(Characterization (23) can be weakened without loss of generality so that the universal quantifications over *relations* X and R are restricted to universal quantifications over *functions* X and R . This, in essence, is what Bird and De Moor [4] refer to as the Eilenberg-Wright lemma.)

[6] Here and elsewhere we use the section notation ($A\otimes$) for the relator $\otimes(K_A \,\Delta\, Id)$.

Exercise 8. The steps to show that T is a relator and catamorphisms are parametric are as follows.

1. Show that in is an isomorphism. The trick is to construct R and S so that $(\!|F\,;R|\!)=\mathsf{id}_A$ and $(\!|F\,;S|\!)\circ\mathsf{in} = (\!|F\,;R|\!)$. (For those familiar with initial algebras in category theory, this is a standard construction.)
2. Now observe that $(\!|F\,;R|\!)$ is the unique solution of the equation

$$X:: \quad X \;=\; R\circ F.X\circ\mathsf{in}^{\cup} \quad .$$

By Knaster-Tarski, it is therefore the least solution of the equation

$$X:: \quad R\circ F.X\circ\mathsf{in}^{\cup}\;\subseteq\; X$$

and the greatest solution of the equation

$$X:: \quad X\;\subseteq\; R\circ F.X\circ\mathsf{in}^{\cup} \quad .$$

(This relies on the assumption that the allegory is locally complete.) Use these two properties to derive (by mutual inclusion) the fusion property:

$$R\circ(\!|F\,;S|\!) \;=\; (\!|F\,;T|\!) \quad\Leftarrow\quad R\circ S \;=\; T\circ F.R \quad .$$

3. Hence derive the fusion law

$$(\!|A\otimes\,;\,R|\!)\circ T.S \;=\; (\!|A\otimes;\; R\circ S\otimes\mathsf{id}|\!)$$

from which (and earlier properties) the fact that T is a relator and catamorphisms are parametric can be verified.

Complete this exercise. □

5.3 Natural Transformations

Reynolds' characterisation of parametric polymorphism predicts that certain polymorphic functions are natural transformations. The point-free formulation of the definition of the arrow operator (14) helps to see this.

Consider, for example, the reverse function on lists, denoted here by rev . This has polymorphic type $\langle\forall\alpha :: \mathsf{List}.\alpha\leftarrow\mathsf{List}.\alpha\rangle$. So, it being parametric is the statement

$$(\mathsf{rev}_A,\mathsf{rev}_B)\in\mathsf{List}.R\leftarrow\mathsf{List}.R$$

for all relations $R : A\leftarrow B$. That is,

$$\mathsf{List}.R\circ\mathsf{rev}_B \;\supseteq\; \mathsf{rev}_A\circ\mathsf{List}.R$$

for all relations R. Similarly the function that makes a pair out of a single value, here denoted by fork , has type $\langle\forall\alpha :: \alpha\times\alpha\leftarrow\alpha\rangle$. So, it being parametric is the statement

$$R\times R\circ\mathsf{fork}_B \;\supseteq\; \mathsf{fork}_A\circ R$$

for all relations $R : A\leftarrow B$.

The above properties of rev and fork are not natural transformation properties because they assert an inclusion and not an equality; they are sometimes called "lax" natural transformation properties. It so happens that the inclusion in the case of rev can be strengthened to an equality but this is certainly not the case for fork. Nevertheless, in the functional programmer's world being a lax natural transformation between two relators is equivalent to being a natural transformation between two functors as we shall now explain.

Since relators are by definition functors, the standard definition of a natural transformation between relators makes sense. That is to say, we define a collection of relations θ indexed by objects (equivalently, a mapping θ of objects to relations) to be *a natural transformation of type* $F \leftarrow G$, for relators F and G iff

$$F.R \circ \theta_B = \theta_A \circ G.R \quad \text{for each } R : A \leftarrow B \ .$$

However, as illustrated by fork above, many collections of relations are not natural with equality but with an inclusion. That is why we define two other types of natural transformation denoted by $F \hookleftarrow G$ and $F \hookrightarrow G$, respectively. We define:

$$\theta : F \hookleftarrow G \quad = \quad F.R \circ \theta_B \supseteq \theta_A \circ G.R \quad \text{for each } R : A \leftarrow B$$

and

$$\theta : F \hookrightarrow G \quad = \quad F.R \circ \theta_B \subseteq \theta_A \circ G.R \quad \text{for each } R : A \leftarrow B \quad .$$

A relationship between naturality in the allegorical sense and in the categorical sense is given by two lemmas. Recall that relators respect functions, i.e. relators are functors on the sub-category Map. The first lemma states that an allegorical natural transformation is a categorical natural transformation:

$$(F.f \circ \theta_B = \theta_A \circ G.f \quad \text{for each function } f : A \leftarrow B\,) \ \Leftarrow \ \theta : F \hookleftarrow G \quad .$$

The second lemma states the converse; the lemma is valid under the assumption that the source allegory of the relators F and G is tabular:

$$\theta : F \hookleftarrow G \ \Leftarrow \ (F.f \circ \theta_B = \theta_A \circ G.f \quad \text{for each function } f : A \leftarrow B\,) \quad .$$

In the case that all elements of the collection θ are *functions* we thus have:

$$\theta : F \hookleftarrow G \text{ in } \mathcal{A} \ \equiv \ \theta : F \leftarrow G \text{ in } Map(\mathcal{A})$$

where by "in X" we mean that all quantifications in the definition of the type of natural transformation range over the objects and arrows of X.

Exercise 9. Prove the above lemmas. □

Since natural transformations of type $F \hookleftarrow G$ are the more common ones and, as argued above, agree with the categorical notion of natural transformation in the case that they are functions, we say that θ is a *natural transformation* if $\theta : F \hookleftarrow G$ and we say that θ is a *proper* natural transformation if $\theta : F \leftarrow G$.

(As mentioned earlier, other authors use the term “lax natural transformation” instead of our natural transformation.)

The natural transformations studied in the computing science literature are predominantly (collections of) functions. In contrast, the natural transformations discussed here are almost all non-functional either because they are partial or because they are non-deterministic (or both).

The notion of arity is of course applicable to all functions defined on product allegories; in particular natural transformations have an arity. A natural transformation of arity $k \leftarrow l$ maps an l-tuple of objects to a k-tuple of relations. The governing rule is: if θ is a natural transformation to F from G (of whatever type — proper or not) then the arities of F and G and θ must be identical. Moreover, the composition $\theta{\circ}\kappa$ of two natural transformations (defined by $(\theta{\circ}\kappa)_A = \theta_A \circ \kappa_A$) is only valid if θ and κ have the same arity (since the composition is componentwise composition in the product allegory).

5.4 Membership and Fans

Since our goal is to use naturality properties to specify relations it is useful to be able to interpret what it means to be “natural”. All interpretations of naturality that we know of assume either implicitly or explicitly that a datatype is a way of structuring information and, thus, that one can always talk about the information stored in an instance of the datatype. A natural transformation is then interpreted as a transformation of one type of structure to another type of structure that rearranges the stored information in some way but does no actual computations on the stored information. Doing no computations on the stored information guarantees that the transformation is independent of the stored information and thus also of the representation used when storing the information.

Hoogendijk and De Moor have made this precise [7]. Their argument, briefly summarised here, is based on the thesis that a datatype is a relator with a membership relation.

Suppose F is an endorelator. The interpretation of $F.R$ is a relation between F-structures of the same shape such that corresponding values stored in the two structures are related by R. For example, $\mathsf{List}.R$ is a relation between two lists of the same length —the shape of a list is its length— such that the ith element of the one list is related by R to the ith element of the other. Suppose A is an object and suppose $X \subseteq \mathsf{id}_A$. So X is a partial identity relation; in effect X selects a subset of A, those values standing in the relation X to themselves. By the same token, $F.X$ is the partial identity relation that selects all F-structures in which all the stored values are members of the subset selected by X. This informal reasoning is the basis of the definition of a membership relation for the datatype F.

The precise specification of membership for F is a collection of relations mem (indexed by objects of the source allegory of F) such that $\mathsf{mem}_A : A \leftarrow F.A$ and such that $F.X$ is the largest subset Y of $\mathsf{id}_{F.A}$ whose “members” are elements of the set X. Formally, mem is required to satisfy the property:

$$\forall(X,Y\colon\ X\subseteq \mathsf{id}_A \wedge Y\subseteq \mathsf{id}_{F.A}\colon\ F.X \supseteq Y \ \equiv\ (\mathsf{mem}_A{\circ}Y)\ \subseteq X) \tag{24}$$

Note that (24) is a Galois connection. A consequence is that a necessary condition for relator F to have membership is that it preserve arbitrary intersections of partial identities. In [7] an example due to P.J. Freyd is presented of a relator that does not have this property. Thus, if one agrees that having membership is an essential attribute of a datatype, the conclusion is that not all relators are datatypes.

Property (24) doesn't make sense in the case that F is not an endorelator but the problem is easily rectified. The general case that we have to consider is a relator of arity $k \leftarrow l$ for some numbers k and l. We consider first the case that k is 1; for $k > 1$ the essential idea is to split the relator into l component relators each of arity $1 \leftarrow k$. For illustrative purposes we outline the case that $l = 2$.

The interpretation of a binary relator $\otimes$ as a datatype-former is that a structure of type $A_0{\otimes}A_1$, for objects A_0 and A_1, contains data at two places: the left and right argument. In other words, the membership relation for $\otimes$ has two components, $\mathsf{mem}_0 : A_0 \leftarrow A_0{\otimes}A_1$ and $\mathsf{mem}_1 : A_1 \leftarrow A_0{\otimes}A_1$, one for each argument. Just as in the endo case, for all $\otimes$-structures being elements of the set $X_0{\otimes}X_1$, for partial identities X_0 and X_1, the component for the left argument should return all and only elements of X_0, the component for the right argument all and only elements of X_1. Formally, we demand that, for all partial identities $X_0 \subseteq \mathsf{id}_{A_0}$, $X_1 \subseteq \mathsf{id}_{A_1}$ and $Y \subseteq \mathsf{id}_{A_0 \otimes A_1}$,

$$X_0{\otimes}X_1 \supseteq Y \ \equiv\ (\mathsf{mem}_0{\circ}Y)\ \subseteq X_0 \ \wedge\ (\mathsf{mem}_1{\circ}Y)\ \subseteq X_1 \tag{25}$$

The rhs of (25) can be rewritten as

$$((\mathsf{mem}_0\,,\mathsf{mem}_1)\circ \Delta_2 Y)\ \subseteq\ (X_0\,,X_1)$$

where Δ_2 denotes the doubling relator: $\Delta_2 Y = (Y\,,Y)$. Now, writing mem in place of ($\mathsf{mem}_0\,,\mathsf{mem}_1$), A in place of ($A_0\,,A_1$) and X in place of ($X_0\,,X_1$), equation (25) becomes, for all partial identities $X \subseteq \mathsf{id}_A$ and $Y \subseteq \mathsf{id}_{(\otimes)A}$,

$$(\otimes)X \supseteq Y \ \equiv\ (\mathsf{mem}\circ \Delta_2 Y)\ \subseteq X \ .$$

In fact, in [7] (24) is not used as the defining property of membership. Instead the following definition is used, and it is shown that (24) is a consequence thereof. (As here, [7] only considers the case $l = 1$.)

Arrow mem is a *membership* relation of endorelator F, if for object A

$$\mathsf{mem}_A\ :\ A \leftarrow F.A$$

and for each pair of objects A and B and each $R : A \leftarrow B$,

$$F.R \circ \mathsf{mem}_B\backslash \mathsf{id}_B = \mathsf{mem}_A\backslash R \ . \tag{26}$$

Properties (26) and (24) are equivalent under the assumption of extensionality as shown by Hoogendijk [8].

Property (26) gives a great deal of insight into the nature of natural transformations. First, the property is easily generalised to:

$$F.R \circ \mathsf{mem}_B \backslash S = \mathsf{mem}_A \backslash (R \circ S) \tag{27}$$

for all $R : A \leftarrow B$ and $S : B \leftarrow C$. Then, it is straightforward to show that the membership, mem , of relator $F : k \leftarrow l$ is a natural transformation. Indeed

$$\mathsf{mem} : Id \hookleftarrow F \quad , \tag{28}$$

and also

$$\mathsf{mem} \backslash \mathsf{id} : F \hookleftarrow Id \quad . \tag{29}$$

Having established these two properties, the —highly significant— observation that mem and $\mathsf{mem} \backslash \mathsf{id}$ are the *largest* natural transformations of their types can be made. Finally, and most significantly, suppose F and G are relators with memberships $\mathsf{mem}.F$ and $\mathsf{mem}.G$ respectively. Then the largest natural transformation of type $F \hookleftarrow G$ is $\mathsf{mem}.F \backslash \mathsf{mem}.G$.

Theorem 1. A polymorphic relation mem is called the *membership* relation of relator F if it has type $\langle \forall \alpha :: \alpha \leftarrow F.\alpha \rangle$ and satisfies (26), for all R . The relation $\mathsf{mem} \backslash \mathsf{id}$ is called the *fan* of F .

Assuming that id is the largest natural transformation of type $Id \hookleftarrow Id$, $\mathsf{mem} \backslash \mathsf{id}$ is the largest natural transformation of type $F \hookleftarrow Id$.

Suppose F and G are relators with memberships $\mathsf{mem}.F$ and $\mathsf{mem}.G$ respectively. Then, $\mathsf{mem}.F \backslash \mathsf{mem}.G$ is the largest natural transformation of type $F \hookleftarrow G$. (In particular, $\mathsf{mem}.F$ is the largest natural transformation of type $Id \hookleftarrow F$.)

Proof. As mentioned, the proofs of (28) and (29) are left as exercises for the reader. (See below.)

Now,

$$\begin{array}{ll}
 & \langle \forall A :: \theta_A \subseteq \mathsf{mem}_A \backslash \mathsf{id}_A \rangle \\
= & \quad \{ \quad \text{factors} \quad \} \\
 & \langle \forall A :: \mathsf{mem}_A \circ \theta_A \subseteq \mathsf{id}_A \rangle \\
\Leftarrow & \quad \{ \quad \text{identification axiom: } \mathsf{id} \text{ is the largest} \\
 & \qquad \text{natural transformation of type } Id \hookleftarrow Id \quad \} \\
 & \mathsf{mem} \circ \theta : Id \hookleftarrow Id \\
\Leftarrow & \quad \{ \quad \text{typing rule for natural transformations} \quad \} \\
 & \mathsf{mem} : Id \hookleftarrow F \ \wedge \ \theta : F \hookleftarrow Id \\
= & \quad \{ \quad (28) \quad \} \\
 & \theta : F \hookleftarrow Id \quad .
\end{array}$$

We thus conclude that $\mathsf{mem} \backslash \mathsf{id}$ is the largest natural transformation of type $F \hookleftarrow Id$.

Now we show that mem is the largest natural transformation of its type. Suppose $\theta : Id \hookleftarrow F$. Then

$$\begin{array}{ll}
 & \theta \\
\subseteq & \{ \quad \text{factors} \quad \} \\
 & \theta \circ \mathsf{mem}\backslash\mathsf{mem} \\
= & \{ \quad (26) \quad \} \\
 & \theta \circ F.\mathsf{mem} \circ \mathsf{mem}\backslash\mathsf{id} \\
\subseteq & \{ \quad \theta : Id \hookleftarrow F \quad \} \\
 & \mathsf{mem} \circ \theta \circ \mathsf{mem}\backslash\mathsf{id} \\
\subseteq & \{ \quad \theta : Id \hookleftarrow F \text{ and } \mathsf{mem}\backslash\mathsf{id} : F \hookleftarrow Id\,. \\
 & \quad\ \text{So } \theta \circ \mathsf{mem}\backslash\mathsf{id} : Id \hookleftarrow Id\,. \\
 & \quad\ \text{Identification axiom.} \quad \} \\
 & \mathsf{mem}\ .
\end{array}$$

Finally, we show that $\mathsf{mem}.F\backslash\mathsf{mem}.G$ is the largest natural transformation of its type. Suppose $\theta : F \hookleftarrow G$. Then,

$$\begin{array}{ll}
 & \mathsf{mem}.F\backslash\mathsf{mem}.G \supseteq \theta \\
= & \{ \quad \text{factors} \quad \} \\
 & \mathsf{mem}.G \supseteq \mathsf{mem}.F \circ \theta \\
\Leftarrow & \{ \quad \mathsf{mem}.G \text{ is the largest natural transformation} \\
 & \quad\ \text{of its type} \quad \} \\
 & \mathsf{mem}.F \circ \theta : Id \hookleftarrow G \\
= & \{ \quad \mathsf{mem}.F : Id \hookleftarrow F \text{ and } \theta : F \hookleftarrow G \\
 & \quad\ \text{composition of natrual transformations} \quad \} \\
 & \mathsf{true}\ .
\end{array}$$

□

Exercise 10. Verify (28) and (29). □

The insight that these properties give is that natural transformations between datatypes can only rearrange values; computation on the stored values or invention of new values is prohibited. To see this, let us consider each of the properties in turn. A natural transformation of type $Id \hookleftarrow F$ constructs values of type A given a structure of type $F.A$. The fact that the membership relation for F is the largest natural transformation of type $Id \hookleftarrow F$ says that all values created by such a natural transformation must be members of the structure $F.A$. Similarly, a natural transformation θ of type $F \hookleftarrow G$ constructs values of type $F.A$ given a structure of type $G.A$. The fact that $\mathsf{mem}.F\backslash\mathsf{mem}.G$ is the largest natural transformation of type $F \hookleftarrow G$ means that every member of the F-structure created by θ is a member of the input G-structure. A proper natural transformation $\theta : F \leftarrow G$ has types $F \hookleftarrow G$ and $F \hookrightarrow G$. So θ has type $F \hookleftarrow G$ and $\theta^{\cup}$ has type $G \hookleftarrow F$. Consequently, a proper natural transformation copies values without loss or duplication.

The natural transformation $\mathsf{mem}\backslash\mathsf{id}$, the largest natural transformation of type $F \hookleftarrow Id$, is called the *canonical fan* of F. It transforms an arbitrary value into an F-structure by non-deterministically creating an F-structure and then copying the given value at all places in the structure. It plays a crucial role in the sequel. (The name "fan" is chosen to suggest the hand-held device that was used in olden times by dignified ladies to cool themselves down.) Rules for computing the canonical fan for all regular relators are as follows. (These are used later in the construction of "zips".)

$$\mathsf{fan}.Proj = \mathsf{id} \tag{30}$$

$$\mathsf{fan}.(F\Delta G) = \mathsf{fan}.F \;\Delta\; \mathsf{fan}.G \tag{31}$$

$$\mathsf{fan}.(F{\cdot}G) = F.(\mathsf{fan}.G) \circ \mathsf{fan}.F \tag{32}$$

$$\mathsf{fan}.K_A = \top\!\!\top_{A,\_} \tag{33}$$

$$\mathsf{fan}.+ = (\mathsf{id}\triangledown\mathsf{id})^{\cup} \tag{34}$$

$$\mathsf{fan}.\times = \mathsf{id}\vartriangle\mathsf{id} \tag{35}$$

$$\mathsf{fan}.T = (\!|\mathsf{id}\otimes\,;\,\mathsf{fan}.\otimes^{\cup}|\!)^{\cup} \tag{36}$$

(where T is the tree relator induced by $\otimes$).

6 Commuting Datatypes — Formal Specification

In this section we formulate precisely what we mean by two datatypes commuting.

Looking again at the examples above, the first step towards an abstract problem specification is clear enough. Replacing "list", "tree" etc. by "datatype F" the problem is to specify an operation $\mathsf{zip}.F.G$ for given datatypes F and G that maps $F{\cdot}G$-structures into $G{\cdot}F$-structures.

We consider only endo relators (relators of arity $1{\leftarrow}1$) here. For a full account, see [8, 6].

The first step may be obvious enough, subsequent steps are less obvious. The nature of our requirements is influenced by the relationship between parametric polymorphism and naturality properties discussed earlier but takes place at a higher level. We consider the datatype F to be fixed and specify a collection of operations $\mathsf{zip}.F.G$ indexed by the datatype G. (The fact that the index is a datatype rather than a type is what we mean by "at a higher level".) Such a family forms what we call a collection of "half-zips". The requirement is that the collection be "parametric" in G. That is, the elements of the family $\mathsf{zip}.F$

should be “logically related” to each other. The precise formulation of this idea leads us to three requirements on “half-zips”. The symmetry between F and G, lost in the process of fixing F and varying G, is then restored by the simple requirement that a zip is both a half-zip and the converse of a half-zip.

The division of our requirements into “half-zips” and “zips” corresponds to the way that zips are constructed. Specifically, we construct a half-zip $\mathsf{zip}.F.G$ for each datatype F in the class of regular datatypes and an arbitrary datatype G. That is to say, for each datatype F we construct the function $\mathsf{zip}.F$ on datatypes which, for an arbitrary datatype G, gives the corresponding zip operation $\mathsf{zip}.F.G$. The function is constructed to meet the requirement that it define a collection of half-zips; subsequently we show that if the collection is restricted to regular datatypes G then each half-zip is in fact a zip.

A further subdivision of the requirements is into naturality requirements and requirements that guarantee that the algebraic structure of pointwise definition of relators is respected (for example, the associativity of functional composition of relators is respected). These we discuss in turn.

6.1 Naturality Requirements

Our first requirement is that $\mathsf{zip}.F.G$ be natural. That is to say, its application to an $F{\cdot}G$-structure should not in any way depend on the values in that structure. Suppose that F and G are relators. Then we demand that

$$\mathsf{zip}.F.G \;:\; G{\cdot}F \leftarrow F{\cdot}G \quad . \tag{37}$$

Thus a zip is a *proper* natural transformation. It transforms one structure to another without loss or duplication of values.

Demanding naturality is not enough. Somehow we want to express that all the members of the family $\mathsf{zip}.F$ of zip operations for different datatypes G and H are related. For instance, if we have a natural transformation $\theta : G \hookleftarrow H$ then $\mathsf{zip}.F.G$ and $\mathsf{zip}.F.H$ should be “coherent” with the transformation θ. That is to say, having both zips and θ, there are two ways of transforming $F{\cdot}H$-structures into $G{\cdot}F$-structures; these should effectively be the same.

One way is first transforming an $F{\cdot}H$-structure into an $F{\cdot}G$-structure using $F{\cdot}\theta$, (i.e. applying the transformation θ to each H-structure inside the F-structure) and then commuting the $F{\cdot}G$-structure into a $G{\cdot}F$-structure using $\mathsf{zip}.F.G$.

Another way is first commuting an $F{\cdot}H$-structure into an $H{\cdot}F$-structure with $\mathsf{zip}.F.H$ and then transforming this H-structure into a G-structure (both containing F-structures) using $\theta{\cdot}F$. So, we have the following diagram.

$$\begin{tikzcd}
F{\cdot}G \arrow[d, "\mathsf{zip}.F.G"'] & F{\cdot}H \arrow[l, "F{\cdot}\theta"'] \arrow[d, "\mathsf{zip}.F.H"] \\
G{\cdot}F & H{\cdot}F \arrow[l, "\theta{\cdot}F"]
\end{tikzcd}$$

One might suppose that an equality is required, i.e.

$$(\theta{\cdot}F)\circ \mathsf{zip}.F.H \quad = \quad \mathsf{zip}.F.G \circ (F{\cdot}\theta) \tag{38}$$

for all natural transformations $\theta : G \hookleftarrow H$. But this requirement is too severe for two reasons.

The first reason is that if θ is not functional, i.e. θ is a non-deterministic transformation, the rhs of equation (38) may be more non-deterministic than the lhs because of the possible multiple occurrence of θ . Take for instance $F := \mathsf{List}$ and $G = H := \times$, i.e. $\mathsf{zip}.F.G$ and $\mathsf{zip}.F.H$ are both the inverse of the zip function on a pair of lists, and take $\theta := \mathsf{id} \cup \mathsf{swap}$, i.e. θ non-deterministically swaps the elements of a pair or not. Then $(\theta{\cdot}F)\circ \mathsf{zip}.F.H$ unzips a list of pairs into a pair of lists and swaps the lists or not. On the other hand, $\mathsf{zip}.F.G \circ (F{\cdot}\theta)$ first swaps some of the elements of a list of pairs and then unzips it into a pair of lists.

The second reason is that, due to the partiality of zips, the domain of the left side of (38) may be smaller than that of the right.

As a concrete example, suppose $\mathsf{listify}$ is a polymorphic function that constructs a list of the elements stored in a tree. The way that the tree is traversed (inorder, preorder etc.) is immaterial; what is important is that $\mathsf{listify}$ is a natural transformation of type $\mathsf{List} \leftarrow \mathsf{Tree}$. Now suppose we are given a list of trees. Then it can be transformed to a list of lists by "listify"ing each tree in the list, i.e. by applying the (appropriate instance of the) function $\mathsf{List.listify}$. If all the trees in the list have the same shape, a list of lists can also be obtained by first commuting the list of trees to a tree of lists (all of the same length) and then "listify"ing the tree structure. That is, we apply the (appropriate instance of the) function $(\mathsf{listify}{\cdot}\mathsf{List})\circ \mathsf{zip.List.Tree}$. The two lists of lists will not be the same: if the size of the original list is m and the size of each tree in the list is n then the first method will construct m lists each of length n whilst the second method will construct n lists each of length m . However the two lists of lists are "zips" of each other ("transposes" would be the more conventional terminology). This is expressed by the commutativity of the following diagram in the case that the input type $\mathsf{List}.(\mathsf{Tree}.A)$ is restricted to lists of trees of the same shape.

$$\begin{tikzcd}
\mathsf{List}.(\mathsf{List}.A) \arrow[d, "(\mathsf{zip.List.List})_A"'] & \mathsf{List}.(\mathsf{Tree}.A) \arrow[l, "\mathsf{List}.(\mathsf{listify}_A)"'] \arrow[d, "(\mathsf{zip.List.Tree})_A"] \\
\mathsf{List}.(\mathsf{List}.A) & \mathsf{Tree}.(\mathsf{List}.A) \arrow[l, "\mathsf{listify}_{\mathsf{List}.A}"]
\end{tikzcd}$$

Note however that if we view both paths through the diagram as partial relations of type $\mathsf{List}.(\mathsf{List}.A) \leftarrow \mathsf{List}.(\mathsf{Tree}.A)$ then the upper path (via $\mathsf{List}.(\mathsf{List}.A)$) includes the lower path (via $\mathsf{Tree}.(\mathsf{List}.A)$). This is because the function $\mathsf{List}.(\mathsf{listify}_A)$ may construct a list of lists all of the same length (as required by the subsequent zip operation) even though all the trees in the given list of trees may not all have the same shape. The requirement on the trees is that they all have the same size, which is weaker than their all having the same shape.

Both examples show that we have to relax requirement (38) using an inclusion instead of equality. Having this inclusion, the requirement for θ can be relaxed as well. So, the requirement becomes

$$(\theta{\cdot}F)\circ\mathsf{zip}.F.H \subseteq \mathsf{zip}.F.G\circ(F{\cdot}\theta) \quad \text{for all } \theta : G \hookleftarrow H \quad . \tag{39}$$

6.2 Composition

For our final requirement we consider the monoid structure of relators under composition. Fix relator F and consider the collection of zips, $\mathsf{zip}.F.G$, indexed by (endo)relator G. Since the (endo)relators form a monoid it is required that the mapping $\mathsf{zip}.F$ is a monoid homomorphism.

In order to formulate this requirement precisely we let ourselves be driven by type considerations. The requirement is that $\mathsf{zip}.F.(G{\cdot}H)$ be some composition of $\mathsf{zip}.F.G$ and $\mathsf{zip}.F.H$ of which $\mathsf{zip}.F.Id$ is the identity. But the type of $\mathsf{zip}.F.(G{\cdot}H)$,

$$\mathsf{zip}.F.(G{\cdot}H) \ : \ G{\cdot}H{\cdot}F \leftarrow F{\cdot}G{\cdot}H \quad ,$$

demands that the datatype F has to be “pushed” through $G{\cdot}H$ leaving the order of G and H unchanged. With $\mathsf{zip}.F.G$ we can swap the order of F and G, with $\mathsf{zip}.F.H$ the order of F and H. Thus transforming $F{\cdot}G{\cdot}H$ to $G{\cdot}H{\cdot}F$ can be achieved as shown below.

$$\begin{tikzcd} G{\cdot}H{\cdot}F & G{\cdot}F{\cdot}H \arrow[l, "G\cdot\mathsf{zip}.F.H"'] & F{\cdot}G{\cdot}H \arrow[l, "(\mathsf{zip}.F.G)\cdot H"'] \end{tikzcd}$$

So, we demand that

$$\mathsf{zip}.F.(G{\cdot}H) \ = \ (G\cdot\mathsf{zip}.F.H)\circ(\mathsf{zip}.F.G\cdot H) \quad . \tag{40}$$

Moreover, in order to guarantee that $\mathsf{zip}.F.(G{\cdot}Id) = \mathsf{zip}.F.G = \mathsf{zip}.F.(Id{\cdot}G)$ we require that

$$\mathsf{zip}.F.Id = \mathsf{id}{\cdot}F \quad . \tag{41}$$

6.3 Half Zips and Commuting Relators

Apart from the very first of our requirements ((37), the requirement that $\mathsf{zip}.F.G$ be natural), all the other requirements have been requirements on the nature of the mapping $\mathsf{zip}.F$. Roughly speaking, (39) demands that it be parametric, and (40) and (41) that it be functorial. We find it useful to bundle the requirements together into the definition of something that we call a “half zip”.

Definition 5 (Half Zip). Consider a fixed relator F and a pointwise closed class of relators $\mathcal{G}$. Then the members of the collection $\mathsf{zip}.F.G$, where G ranges over $\mathcal{G}$, are called *half-zips* iff

(a) $\mathsf{zip}.F.G \ : \ G{\cdot}F \leftarrow F{\cdot}G$, for each G ,
(b) $\mathsf{zip}.F.(G{\cdot}H) \ = \ (G \cdot \mathsf{zip}.F.H) \circ (\mathsf{zip}.F.G \cdot H)$ for all G and H ,
(c) $(\theta{\cdot}F) \circ \mathsf{zip}.F.H \ \subseteq \ \mathsf{zip}.F.G \circ (F{\cdot}\theta)$ for each $\theta : G \hookleftarrow H$.

(In addition, zips should respect the pointwise closure of the class $\mathcal{G}$, but this aspect of the definition is omitted here.) □

Definition 6 (Commuting Relators). The half-zip $\mathsf{zip}.F.G$ is said to be a *zip* of (F , G) if there exists a half-zip $\mathsf{zip}.G.F$ such that

$$\mathsf{zip}.F.G \ = \ (\mathsf{zip}.G.F)^{\cup}$$

We say that datatypes F and G *commute* if there exists a zip for (F , G). □

7 Consequences

In this section we address two concerns. First, it may be the case that our requirement is so weak that it has many trivial solutions. We show that, on the contrary, the requirement has a number of consequences that guarantee that there are no trivial solutions. On the other hand, it could be that our requirement for datatypes to commute is so strong that it is rarely satisfied. Here we show that the requirement can be met for all regular datatypes. (Recall that the "regular" datatypes are the sort of datatypes that one can define in a conventional functional programming language.) Moreover, we can even prove the remarkable result that for the regular relators our requirement has a *unique* solution.

7.1 Shape Preservation

Zips are partial operations: $\mathsf{zip}.F.G$ should map F –structures of (G –structures of the same shape) into G –structures of (F –structures of the same shape). This requirement is, however, not explicitly stated in our formalisation of being a zip. In this subsection we show that it is nevertheless a consequence of that formal requirement. In particular we show that a half zip always constructs G –structures of (F –structures of the same shape). We in fact show a more general result that forms the basis of the uniqueness result for regular relators.

Let us first recall how shape considerations are expressed. The function $!_A$ is the function of type $\mathbb{1} \leftarrow A$ that replaces a value by the unique element of the unit type, $\mathbb{1}$. Also, for an arbitrary function f , $F.f$ maps an F –structure to an F –structure of the same shape, replacing each value in the input structure by the result of applying f to that value. Thus $F.!_A$ maps an F –structure (of A 's) to an F –structure of the same shape in which each value in the input structure has been replaced by the unique element of the unit type. We can say that $(F.!_A).x$ *is* the shape of the F –structure x , and $F.!_A \circ f$ *is* the shape of the result of applying function f .

Now, for a natural transformation θ of type $F \leftarrow G$, the shape characteristics of α in general are determined by $\theta_{\mathbb{1}}$, since

$$F.!_A \circ \theta_A = \theta_{\mathbb{1}} \circ G.!_A$$

That is, the shape of the result of applying θ_A is completely determined by the behaviour of $\theta_{\mathbb{1}}$. The shape characteristics of $\mathsf{zip}.F.G$, in particular, are determined by $(\mathsf{zip}.F.G)_{\mathbb{1}}$ since

$$(G{\cdot}F).!_A \circ (\mathsf{zip}.F.G)_A = (\mathsf{zip}.F.G)_{\mathbb{1}} \circ (F{\cdot}G).!_A$$

Our shape requirement is that a half zip maps an $F-G$–shape into a $G-F$–shape in which all F–shapes equal the original F–shape. This we can express by a single equation relating the behaviour of $(\mathsf{zip}.F.G)_{\mathbb{1}}$ to that of $\mathsf{fan}.G$. Specifically, we note that $(\mathsf{fan}.G)_{F.\mathbb{1}}$ generates from a given F-shape, x , an arbitrary G-structure in which all elements equal x , and thus have the same F–shape. On the other hand, $F.((\mathsf{fan}.G)_{\mathbb{1}})$, when applied to x , generates F-structures with shape x containing arbitrary G-shapes. The shape requirement (for endorelators) is thus satisfied if we can establish the property

$$(\mathsf{fan}.G)_{F.\mathbb{1}} \;=\; (\mathsf{zip}.F.G)_{\mathbb{1}} \circ F.((\mathsf{fan}.G)_{\mathbb{1}}) \;\; . \tag{42}$$

This property is an immediate consequence of the following lemma.

Suppose F and G are datatypes. Then, if $\mathsf{fan}.G$ is the canonical fan of G ,

$$\mathsf{fan}.G \cdot F \;=\; \mathsf{zip}.F.G \circ (F \cdot \mathsf{fan}.G) \;\; . \tag{43}$$

From equation (42) it also follows that the range of $(\mathsf{zip}.F.G)_{\mathbb{1}}$ is the range of $(\mathsf{fan}.G)_{F.\mathbb{1}}$, i.e. arbitrary G-structures of which all elements are the same, but arbitrary, F-shape.

A more general version of (43) is obtained by considering the so-called *fan function*. Recalling the characterising property of the membership relation (26), we define the mapping $\hat{F}$ (with the same arity as F , namely $k \leftarrow l$) by

$$\hat{F}.R = F.R \circ \mathsf{mem}\backslash\mathsf{id} \;\; , \tag{44}$$

for all $R : A \leftarrow B$. Then the generalisation of (43) is the following lemma.

Suppose F and G are datatypes. Then, if $\hat{G}$ is the fan function of G ,

$$(\hat{G}{\cdot}F).R = (\mathsf{zip}.F.G)_A \circ (F{\cdot}\hat{G}).R \;\; , \tag{45}$$

for all $R : A \leftarrow B$.

It is (45) that often uniquely characterises $\mathsf{zip}.F.G$. (In fact, our initial study of zips [3] was where the notion of a fan was first introduced, and (45) was proposed as one of the defining properties of a zip.)

7.2 All Regular Datatypes Commute

We conclude this discussion by showing that all regular relators commute. Morever, for each pair of regular relators F and G there is a *unique* natural transformation $\mathsf{zip}.F.G$ satisfying our requirements.

The regular relators are constructed from the constant relators, product and coproduct by pointwise extension and/or the construction of tree relators. The requirement that $\mathsf{zip}.F.G$ and $\mathsf{zip}.G.F$ be each other's converse (modulo transposition) demands the following definitions:

$$\mathsf{zip}.Id.G = \mathsf{id}{\cdot}G \quad , \tag{46}$$

$$\mathsf{zip}.Proj.G = \mathsf{id} \quad , \tag{47}$$

$$\mathsf{zip}.(F\,\varDelta\,G).H = (\mathsf{zip}.F.H\,,\,\mathsf{zip}.G.H) \quad , \tag{48}$$

$$\mathsf{zip}.(F{\cdot}G).H = (\mathsf{zip}.F.H\cdot G)\circ(F\cdot\mathsf{zip}.G.H) \quad . \tag{49}$$

For the constant relators and product and coproduct, the zip function is uniquely characterised by (45). One obtains the following definitions, for all G:

$$\mathsf{zip}.K_A.G = \mathsf{fan}.G\cdot K_A \quad , \tag{50}$$

$$\mathsf{zip}.+.G = G.\mathsf{inl}\triangledown G.\mathsf{inr} \quad , \tag{51}$$

$$\mathsf{zip}.\times.G = (G.\mathsf{outl}\vartriangle G.\mathsf{outr})^{\cup} \quad . \tag{52}$$

Note that, in general, $\mathsf{zip}.K_A.G$ and $\mathsf{zip}.\times.G$ are not simple; moreover, the latter is typically partial. That is the right domain of $(\mathsf{zip}.\times.G)_{(A,B)}$ is typically a proper subset of $G.A\times G.B$. Zips of datatypes defined in terms of these datatypes will thus also be non-simple and/or partial. Nevertheless, broadcast operations are always functional.

Tree relators are the last sort of relators in the class of regular relators. Let T be the tree relator induced by $\otimes$ as defined in section 5.2. Then,

$$\mathsf{zip}.T.G = (\!|\mathsf{id}_G\otimes\,;\;G.\mathsf{in}\circ(\mathsf{zip}.\otimes.G\,\cdot\,(Id\,\varDelta\,T))|\!) \quad . \tag{53}$$

8 Conclusion

The purpose of these lectures has been to introduce and study datatype-generic programs, emphasising genericity of specifications and proof. A relational theory of datatypes was introduced, motivated by the relational formulation of parametric polymorphism. Central to this theory is a (generic) formulation of the notion of a relator with membership. A fundamental insight, formally expressible within the theory, is that a natural transformation is a structural transformation that rearranges information (but does not create or alter information) possibly replicating or losing information in the process. A proper natural transformation rearranges information without loss or replication.

The problem of "commuting" two datatypes (transforming an F-structure of G-structures into a G-structure of F-structures) has been used to illustrate the effectiveness of a datatype-generic theory of programming. The specification of this problem is based on abstraction from the properties of brodacasting and matrix transposition, and is entirely non-operational. In this way, very general properties of the constructed programs are derived with a minimum of effort.

The development of a constructive theory of datatype genericity is a challenging area of research. Almost all literature on generic programming is about implementations with little or no regard to generic specification or proof. But finding the right abstractions to make the right building blocks demands that specification and proof be given as much consideration as implementation.

References

1. R.C. Backhouse, P. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C.S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 303–326. Springer-Verlag, Workshops in Computing, 1992.
2. R.C. Backhouse, P. de Bruin, G. Malcolm, T.S. Voermans, and J. van der Woude. Relational catamorphisms. In Möller B., editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers B.V., 1991.
3. R.C. Backhouse, H. Doornbos, and P. Hoogendijk. Commuting relators. Available via World-Wide Web at `http://www.cs.nott.ac.uk/~rcb/MPC/papers`, September 1992.
4. Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall International, 1996.
5. P.J. Freyd and A. Ščedrov. *Categories, Allegories.* North-Holland, 1990.
6. Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science, 7th International Conference*, volume 1290 of *LNCS*, pages 242–260. Springer-Verlag, September 1997.
7. Paul Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.
8. Paul Hoogendijk. *A Generic Theory of Datatypes.* PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.
9. R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.
10. R. Milner. The standard ML core language. *Polymorphism*, II(2), October 1985.
11. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
12. J. Riguet. Relations binaires, fermetures, correspondances de Galois. *Bulletin de la Société Mathématique de France*, 76:114–155, 1948.
13. C. Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, August 1967.
14. P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.

Basic Category Theory for Models of Syntax[⋆]

R.L. Crole

Department of Mathematics and Computer Science, University of Leicester, Leicester, LE1 7RH, UK
R.Crole@mcs.le.ac.uk

Abstract. A preliminary version of these notes formed the basis of four lectures given at the *Summer School on Generic Programming*, Oxford, UK, which took place during August 2002. The aims of the notes are to provide an introduction to very elementary category theory, and to show how such category theory can be used to provide both abstract and concrete mathematical models of syntax. Much of the material is now standard, but some of the ideas which are used in the modeling of syntax involving variable binding are quite new.
It is assumed that readers are familiar with elementary set theory and discrete mathematics, and have met formal accounts of the kinds of syntax as may be defined using the (recursive) datatype declarations that are common in modern functional programming languages. In particular, we assume readers know the basics of λ-calculus.
A pedagogical feature of these notes is that we only introduce the category theory required to present models of syntax, which are illustrated by example rather than through a general theory of syntax.

1 Introduction

1.1 Prerequisites

We assume that readers already have a reasonable understanding of

– very basic (naive) set theory;
– simple discrete mathematics, such as relations, functions, preordered sets, and equivalence relations;
– simple (naive) logic and the notion of a formal system;
– a programming language, preferably a functional one, and in particular of recursive datatypes; language syntax presented as finite trees;
– inductively defined sets; proof by mathematical and rule induction;
– the basics of λ-calculus, especially free and bound variables.

Some of these topics will be reviewed as we proceed. The appendix defines abstract syntax trees, inductively defined sets, and rule induction. We *do not* assume any knowledge of category theory, introducing all that we need.

⋆ This work was supported by EPSRC grant number GR/M98555.

1.2 The Aims

Through these notes we aim to teach some of the very basics of category theory, and to apply this material to the study of programming language syntax with binding. We give formal definitions of the category theory we need, and some concrete examples. We also give a few technical results which can be given very abstractly, but for the purposes of these notes are given in a simplified and concrete form. We study syntax through particular examples, by giving categorical models. We do not discuss the general theory of binding syntax. This is a current research topic, but readers who study these notes should be well placed to read (some of) the literature.

1.3 Learning Outcomes

By studying these notes, and completing at least some of the exercises, you should

- know how simple examples of programming language syntax with binding can be specified via (simple) inductively defined formal systems;
- be able to define categories, functors, natural transformations, products and coproducts, presheaves and algebras;
- understand a variety of simple examples from basic category theory;
- know, by example, how to compute initial algebras for polynomial functors over sets;
- understand some of the current issues concerning how we model and implement syntax involving variable binding;
- understand some simple abstract models of syntax based on presheaves;
- know, by example, how to manufacture a categorical formulation of simple formal systems (for syntax);
- be able to prove that the abstract model and categorical formal system are essentially the same;
- know enough basic material to read some of the current research material concerning the implementation and modeling of syntax.

2 Syntax Defined from Datatypes

In this section we assume that readers are already acquainted with formal (programming) languages. In particular, we assume knowledge of syntax trees; occurrences of, free, bound and binding variables in a syntax tree; inductive definitions and proof by (rule) induction; formal systems for specifying judgements about syntax; and elementary facts about recursive datatype declarations. Some of these topics are discussed very briefly in the Appendix, page 170. Mostly, however, we prompt the reader by including definitions, but we do not include extensive background explanations and intuition.

What is the motivation for the work described in these notes? Computer Scientists sometimes want to use existing programming languages, and other tools such as automated theorem provers, to reason about (other) programming languages. We sometimes say that the existing system is our **metalanguage**, (for example Haskell) and the system to be reasoned about is our **object language** (for example, the λ-calculus). Of course, the object language will (hopefully) have a syntax, and we will need to encode, or specify, this object level language within the metalanguage. It turns out that this is not so easy, even for fairly simple object languages, if the object language has binders. This is true of the λ-calculus, and is in fact true of almost any (object level) programming language.

We give four examples, each showing how to specify the syntax, or **terms**, of a tiny fragment of a (object level) programming language. In each case we give a datatype whose elements are called *expressions*. Such expressions denote *abstract syntax trees*—see Appendix. The expressions form a superset of the set of terms we are interested in. We then give *further definitions* which specify a *subset* of the datatype which constitutes exactly the terms of interest. For example, in the case of λ-calculus, the expressions are raw syntax trees, and the terms are such trees quotiented up to α-equivalence. What we would like to be able to do is

> *Write down a datatype, as one can do in a functional programming language, such that the expressions of the datatype are precisely the terms of the object language we wish to encode.*

Then, the "further definitions" alluded to above would not be needed. We would have a very clean encoding of our object level language terms, given by an explicit datatype and no other infrastructure (such as α-equivalence).

2.1 An Example with Distinguished Variables and without Binding

Take **constructor symbols** V, S and A with arities one, one and two respectively. Take also a **set of variables** $\mathbb{V}$ whose *actual* elements will be denoted by v^i where $i \in \mathbb{N}$. The **set of expressions** Exp is inductively defined by the grammar[1]

$$\mathit{Exp} ::= \mathsf{V}\ \mathbb{V} \mid \mathsf{S}\ \mathit{Exp} \mid \mathsf{A}\ \mathit{Exp}\ \mathit{Exp}$$

The metavariable e will range over expressions. It should be intuitively clear what we mean by v^i **occurs** in e, which we abbreviate by $v^i \in e$. We omit a formal definition. The set of **(free) variables** of any e is denoted by $\mathit{fv}(e)$.

Later in these notes, we will want to consider expressions e for which

$$\mathit{fv}(e) \subset \{\, v^0, \dots, v^{n-1} \,\}$$

The idea is that when "constructing" an expression by using the grammar above, we build it out of the initial segment (according to the indices) of variables. It is convenient to give an inductive definition of such expressions. First we inductively a set of judgements $\Gamma^n \vdash^{\mathsf{d\bar{b}}} e$ where $n \geq 1$, $\Gamma^n \stackrel{\text{def}}{=} v^0, \dots, v^{n-1}$ is a list,

[1] This formal BNF grammar corresponds to a datatype declaration in Haskell.

$$\frac{0 \leq i < n}{\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} \mathsf{V}\, v^i} \qquad \frac{\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} e}{\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} \mathsf{S}\, e} \qquad \frac{\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} e \quad \Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} e'}{\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} \mathsf{A}\, e\, e'}$$

Fig. 1. Expressions in Environment—Distinguished Variables, No Binding

$$\frac{0 \leq i < n}{\Gamma^n \vdash^{\mathsf{db}} \mathsf{V}\, v^i} \qquad \frac{\Gamma^{n+1} \vdash^{\mathsf{db}} e}{\Gamma^n \vdash^{\mathsf{db}} \mathsf{L}\, v^n\ e} \qquad \frac{\Gamma^n \vdash^{\mathsf{db}} e \quad \Gamma^n \vdash^{\mathsf{db}} e'}{\Gamma^n \vdash^{\mathsf{db}} \mathsf{E}\, e\, e'}$$

Fig. 2. Expressions in Environment—Distinguished Variables and Binding

and of course e is an expression. We refer to Γ^n as an **environment** of variables. The rules appear in Figure 1. $\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} e$ is simply a notation for a binary relationship, written infix style, with relation symbol $\vdash^{\mathsf{d}\bar{\mathsf{b}}}$. Strictly speaking, we are inductively defining the set (of pairs) $\vdash^{\mathsf{d}\bar{\mathsf{b}}}$. One can then prove by rule induction that if $\Gamma^n \vdash^{\mathsf{d}\bar{\mathsf{b}}} e$ then $fv(e) \subset \Gamma^n$. Check the simple details as an *Exercise. Hint: With the notation of the Appendix, we prove by Rule Induction*

$$(\forall\, (\Gamma^n, e) \in \vdash^{\mathsf{d}\bar{\mathsf{b}}})\ \boxed{(fv(e) \subset \Gamma^n)}$$

See Figure 7, page 173. One has to check property closure for each rule in Figure 1.

2.2 An Example with Distinguished Variables and Binding

We assume that readers are familiar with the syntax of the λ-calculus. One might want to implement (encode) such syntax in a language such as Haskell. If L and E are new constructors then we might consider the grammar (datatype)

$$\mathit{Exp} ::= \mathsf{V}\ \mathbb{V} \mid \mathsf{L}\ \mathbb{V}\ \mathit{Exp} \mid \mathsf{E}\ \mathit{Exp}\ \mathit{Exp}$$

We assume that readers are familiar with **free**, **binding** and **bound** variables, in particular the idea that for a particular occurrence of a variable in a syntax tree defined by the grammar above, the occurrence is either free or bound. In particular recall that $fv(\mathsf{L}\ v^i\ e) = fv(e) \setminus \{\, v^i \,\}$; any occurrence of v^i in $\mathsf{L}\ \underline{v^i}\ e$ is bound, and in particular we call the occurrence $\underline{v^i}$ binding. We will want to consider expressions e for which $fv(e) \subset \{\, v^0, \ldots, v^{n-1} \,\}$, and we also give an inductive definition of such expressions. More precisely we inductively define a set of judgements $\Gamma^n \vdash^{\mathsf{db}} e$ where $n \geq 1$. The rules appear in Figure 2. One can then prove by rule induction that if $\Gamma^n \vdash^{\mathsf{db}} e$ then $fv(e) \subset \Gamma^n$. Check the simple details as an *Exercise.* Notice that the rule for introducing abstractions $\mathsf{L}\ v^n\ e$ forces a **distinguished** choice of binding variable. This means that we lose the usual fact that the name of the binding variable does not matter. However, the pay-off of what we call *distinguished binding* is that the expressions inductively

$$\frac{x \in \Delta}{\Delta \vdash^{\mathsf{ab}} \mathsf{V}\ x} \qquad \frac{\Delta, x \vdash^{\mathsf{ab}} e}{\Delta \vdash^{\mathsf{ab}} \mathsf{L}\ x\ e} \qquad \frac{\Delta \vdash^{\mathsf{ab}} e \quad \Delta \vdash^{\mathsf{ab}} e'}{\Delta \vdash^{\mathsf{ab}} \mathsf{E}\ e\ e'}$$

Fig. 3. Expressions in Environment—Arbitrary Variables and Binding

defined in Figure 2 correspond exactly to the terms of the λ-calculus, without the need to define α-equivalence. In essence, we are forced to pick a representative of each α-equivalence class.

2.3 An Example with Arbitrary Variables and Binding

Expressions are still defined by

$$\mathit{Exp} ::= \mathsf{V}\ \mathbb{V} \mid \mathsf{L}\ \mathbb{V}\ \mathit{Exp} \mid \mathsf{E}\ \mathit{Exp}\ \mathit{Exp}$$

Now let Δ range over *all non-empty finite lists* of variables *which have distinct elements*. Thus a typical non-empty Δ is $v^1, v^8, v^{100}, v^2 \in [\mathbb{V}]$. We will use typical metavariables x, y, z which range over $\mathbb{V}$. If Δ contains $n \geq 1$ variables, we may write $\Delta = x_0, \ldots, x_{n-1}$. Once again we inductively a set of judgements, this time of the form $\Delta \vdash^{\mathsf{ab}} e$. The rules appear in Figure 3. One can then prove by rule induction that if $\Delta \vdash^{\mathsf{ab}} e$ then $\mathit{fv}(e) \subset \Delta$. Check the simple details as an *Exercise.*

It is convenient to introduce a notion of simultaneous substitution of variables at this point. This will allow us to define the usual notion of α-equivalence of expressions—yielding the *terms* of the λ-calculus. Such substitution will also be used in our mathematical models, later on. Suppose that $0 \leq p \leq \mathsf{len}(\Delta) - 1$. Then $\mathsf{el}_p(\Delta)$ is the pth element of Δ, with position 0 the "first" element. We write ϵ for the empty list. We will define by recursion over expressions e, new expressions $e\{\epsilon/\epsilon\}$ and $e\{\Delta'/\Delta\}$, where $\mathsf{len}(\Delta) = \mathsf{len}(\Delta')$. Informally, $e\{\Delta'/\Delta\}$ is the expression e in which any free occurrence of $\mathsf{el}_p(\Delta)$ in e is replaced by $\mathsf{el}_p(\Delta')$, with bound variables being changed to avoid capture of $\mathsf{el}_p(\Delta')$. For example,

$$(\mathsf{L}\ v^8\ (\mathsf{A}\ v^{10}\ v^2))\{v^3, v^8/v^8, v^2\} = \mathsf{L}\ v^{11}\ (\mathsf{A}\ v^{10}\ v^8)$$

where the binding variable v^8 is changed to v^{11}. The definition is given in Figure 4.

We can inductively define the relation $\sim_\alpha$ of **α-equivalence** on the set of all expressions with the single axiom[2] (schema) $\mathsf{L}\ x\ e \sim_\alpha \mathsf{L}\ x'\ e\{x'/x\}$ where x' $(\neq x)$ is any variable not in $\mathit{fv}(e)$, and rules ensuring that $\sim_\alpha$ is an equivalence relation and a congruence for the constructors. Congruence for L, and transitivity, are given by the rules (schemas)

$$\frac{e \sim_\alpha e' \qquad e' \sim_\alpha e''}{e \sim_\alpha e''} \qquad\qquad \frac{e \sim_\alpha e'}{\mathsf{L}\ x\ e \sim_\alpha \mathsf{L}\ x\ e'}$$

[2] Base rule.

We define

$$e\{\epsilon/\epsilon\} \stackrel{\text{def}}{=} e$$

$$(\mathsf{V}\ x)\{\Delta'/\Delta\} \stackrel{\text{def}}{=} \begin{cases} x \\ \quad \text{if} \quad (\forall p)(\mathsf{el}_p(\Delta) \neq x) \\ \mathsf{el}_p(\Delta') \\ \quad \text{if} \quad (\exists p)(\mathsf{el}_p(\Delta) = x) \end{cases}$$

$$(\mathsf{L}\ x\ e)\{\Delta'/\Delta\} \stackrel{\text{def}}{=} \begin{cases} \mathsf{L}\ x\ e\{\overline{\Delta'}/\overline{\Delta}\} \\ \quad \text{if} \quad (\forall p)(\mathsf{el}_p(\Delta') \neq x\ \vee\ \mathsf{el}_p(\Delta) \notin fv(e)) \\ \mathsf{L}\ x'\ e\{\overline{\Delta'}, x'/\overline{\Delta}, x\} \\ \quad \text{if} \quad (\exists p)(\mathsf{el}_p(\Delta') = x \wedge \mathsf{el}_p(\Delta) \in fv(e)) \end{cases}$$

$$(\mathsf{E}\ e\ e')\{\Delta'/\Delta\} \stackrel{\text{def}}{=} \mathsf{E}\ e\{\Delta'/\Delta\}\ e'\{\Delta'/\Delta\}$$

where

– $\overline{\Delta}$ is Δ with x deleted (from position p, if it occurs) and, *if x does occur*, $\overline{\Delta'}$ is Δ' with the element in position p deleted, and is otherwise Δ'; and
– x' is the variable v^w where w is 1 plus the maximum of the indices appearing in $\overline{\Delta'}$ and $fv(e)$.

Fig. 4. Definition of Simultaneous Substitution

Write down all of the rules as an *Exercise.*

Note that the terms of the λ-calculus are given by the

$$[e]_\alpha \stackrel{\text{def}}{=} \{e' \mid e' \sim_\alpha e\}$$

where e is any expression. One reason for also defining the judgements $\Delta \vdash^{\mathsf{ab}} e$ is that they will be used in Section 4 to formulate a mathematical model.

2.4 An Example without Variables but with Binding

As mentioned, we assume familiarity with de Bruijn notation, and the notion of *index level.* **Note: You can follow the majority of these notes, without knowing about de Bruijn terms. Just omit the sections on this topic.** Here is our grammar of raw de Bruijn expressions

$$Exp ::= \mathsf{V}\ \mathbb{N} \mid \lambda\ Exp \mid \$\ Exp\ Exp$$

Recall the basic idea is that the property of syntax which is captured by variable binding, can also be embodied by *node count* in an expression. An example may make this more transparent. Consider

$$e \stackrel{\text{def}}{=} \mathsf{L}\ v^1\ (\mathsf{L}\ v^5\ \mathsf{L}\ v^4\ (\mathsf{E}\ v^5 v^1\ ))$$

$$\frac{0 \leq j \leq n-1}{n \vdash^{\mathsf{ib}} \mathsf{V}\, j} \qquad \frac{n+1 \vdash^{\mathsf{ib}} e}{n \vdash^{\mathsf{ib}} \lambda\, e} \qquad \frac{n \vdash^{\mathsf{ib}} e \quad n \vdash^{\mathsf{ib}} e'}{n \vdash^{\mathsf{ib}} \$\, e\, e'}$$

Fig. 5. Expressions in Environment—Binding via Node Depth

The first occurrence of v^1 (binding) indicates that the second occurrence is bound. However, if we draw out the finite tree, we can see that there is a single path between the two variables, and the fact that the second v^1 is bound by the first is "specified" by counting the number, 2, of nodes L *strictly between* them. There is also one L between the two occurrences of v^5. In de Bruijn notation, e is rendered $\lambda\ (\lambda\ (\lambda\ (\$\ 1\ 2)))$. For example, $\lambda\ 0$ corresponds to $\mathsf{L}\ v^0\ v^0$ and $\mathsf{L}\ v^0\ (\mathsf{L}\ v^1 v^0\ )$ to $\lambda\ (\lambda\ 1)$. Now let n range over $\mathbb{N}$. We will regard n as the set $\{0, \dots, n-1\}$ of natural numbers, with the elements treated as De Bruijn indices. We inductively define a set of judgements, this time of the form $n \vdash^{\mathsf{ib}} e$. The rules appear in Figure 5. One can then prove by rule induction that if $n \vdash^{\mathsf{ib}} e$ then e is a de Bruijn expression of level n. Check the simple details as an *Exercise.*

We finish this section with an *Exercise.* Write down rules of the form $\Delta \vdash e$ with e corresponding to the datatype of Section 2.1, and Δ as defined in Section 2.3. Show that in this example, where variables are arbitrary and there is no binding, the set of expressions e for which $\Delta \vdash e$ is precisely the set of elements of the datatype.

3 Category Theory

3.1 Categories

A category consists of two collections. Elements of one collection are called *objects.* Elements of the other collection are called *morphisms.* Speaking informally, each morphism might be thought of as a "relation" or "connection" between two objects. Here are some informal examples.

• The collection of all sets (each set is an object), together with the collection of all set-theoretic functions (each function is morphism).
• The collection of all posets, together with all monotone functions.
• The set of real numbers $\mathbb{R}$ (in this case each object is just a real number r in $\mathbb{R}$), together with the order relation $\leq$ on the set $\mathbb{R}$ (a morphism is a pair (r, r') in the set $\leq$).

It is important to note that the objects of a category do not have to be sets (in the fourth example they are real numbers) and that the morphisms do not have to be functions (in the fourth example they are elements of the order relation $\leq$). Of course, there are some precise rules which define exactly what a category is, and we come to these shortly: the reader may care to glance ahead at the definition given on page 140. We give a more complete example before coming to the formal definition.

We can create an example of a category using the definitions of Section 2.1. We illustrate carefully the general definition of a category using this example. The collection of *objects* is $\mathbb{N}$ and the collection of *morphisms* is

$$\bigcup_{m \geq 1} [\{ \, e \mid \Gamma^m \vdash^{\mathsf{d\bar{b}}} e \, \}]$$

Given any morphism es, there is a corresponding *source* and *target*. We define

$$src(es) \stackrel{\text{def}}{=} \mathsf{max} \, \{ \, i \mid v^i \in e \wedge e \in es \, \} + 1$$

and

$$tar(es) \stackrel{\text{def}}{=} \; \mathsf{case\ of} \begin{cases} \epsilon \to 0 \\ [\, e_0, \dots, e_{n-1} \,] \to n \end{cases}$$

We write $es{:}\, m \to n$ or $m \overset{es}{\to} n$ to indicate the source and target of es. For example, we have

$$2 \xrightarrow{[\,\mathsf{A}\ (\mathsf{A}\ v^0\ v^0)\ v^1, \mathsf{A}\ v^1\ v^0, \mathsf{A}\ v^0\ (\mathsf{S}\ v^0)\,]} 3$$

In the following situation

$$\begin{tikzcd} l \arrow[r, "es'"] & m \arrow[r, "es"] & n \end{tikzcd}$$

we say that es and es' are *composable*. This means that there is a morphism $es \circ es'{:}\, l \to n$ which can be defined using es and es', and is said to be their composition. For example, if

$$1 \xrightarrow{[\,\mathsf{S}\ v^0, \mathsf{A}\ v^0\ v^0\,]} 2 \xrightarrow{[\,\mathsf{A}\ (\mathsf{A}\ v^0\ v^1)\ v^1, \mathsf{A}\ v^1\ v^0, \mathsf{A}\ v^0\ (\mathsf{S}\ v^1)\,]} 3$$

then the composition is

$$1 \xrightarrow{[\,\mathsf{A}\ (\mathsf{A}\ (\mathsf{S}\ v^0)\ (\mathsf{A}\ v^0\ v^0))\ (\mathsf{A}\ v^0\ v^0), \mathsf{A}\ (\mathsf{A}\ v^0\ v^0)\ (\mathsf{S}\ v^0), \mathsf{A}\ (\mathsf{S}\ v^0)\ (\mathsf{S}\ (\mathsf{A}\ v^0\ v^0))\,]} 3$$

Informally, the general definition of composition is that the element in position p in $es \circ es'$ is the element in es in position p in which the m elements of es' are substituted simultaneously for the free variables $v^0, \dots v^{m-1}$. We leave the actual definition as an *Exercise*—do not forget what happens if either es or es' is empty. Finally, for any object m there is an *identity* morphism,

$$m \xrightarrow{[\, v^0, \dots, v^{m-1} \,]} m$$

We now give the formal definition of a category. A **category** $\mathcal{C}$ is specified by the following data:

• A collection *ob* $\mathcal{C}$ of entities called **objects**. An object will often be denoted by a capital letter such as A, B, $C \dots$

• A collection $mor\ \mathcal{C}$ of entities called **morphisms**. A morphism will often be denoted by a small letter such as f, g, $h\ldots$
• Two operations assigning to each morphism f its **source** $src(f)$ hich is an object of $\mathcal{C}$ and its **target** $tar(f)$ lso an object of $\mathcal{C}$. We write $f\colon src(f) \longrightarrow tar(f)$ to indicate this, or perhaps $f\colon A \to B$ where $A = src(f)$ and $B = tar(f)$. Sometimes we just say "let $f\colon A \to B$ be a morphism of $\mathcal{C}$" to mean f is a morphism of $\mathcal{C}$ with source A and target B.
• Morphisms f and g are **composable** if $tar(f) = src(g)$. There is an operation assigning to each pair of composable morphisms f and g their **composition** which is a morphism denoted by $g \circ f$ and such that $src(g \circ f) = src(f)$ and $tar(g \circ f) = tar(g)$. So for example, if $f\colon A \to B$ and $g\colon B \to C$, then there is a morphism $g \circ f\colon A \to C$. There is also an operation assigning to each object A of $\mathcal{C}$ an **identity** morphism $id_A\colon A \to A$. These operations are required to be **unitary**

$$id_{tar(f)} \circ f = f$$
$$f \circ id_{src(f)} = f$$

and **associative**, that is given morphisms $f\colon A \to B$, $g\colon B \to C$ and $h\colon C \to D$ then

$$(h \circ g) \circ f \;=\; h \circ (g \circ f).$$

As an *Exercise* check that the operations from the previous example are unitary and associative.

Here are some more examples of categories.

1. The category of sets and total functions, $\mathcal{S}et$. The objects of the category are **sets** and the morphisms are **functions** which are triples (A, f, B) where A and B are sets and $f \subseteq A \times B$ is a subset of the cartesian product of A and B for which
$$(\forall a \in A)(\exists! b \in B)((a, b) \in f)$$
We sometimes call f the **graph** of the function (A, f, B). The source and target operations are defined by $src(A, f, B) \stackrel{\text{def}}{=} A$ and $tar(A, f, B) \stackrel{\text{def}}{=} B$. Suppose that we have another morphism (B, g, C). Then $tar(A, f, B) = src(B, g, C)$, and the composition is given by
$$(B, g, C) \circ (A, f, B) = (A, g \circ f, C)$$
where $g \circ f$ is the usual composition of the graphs f and g. Finally, if A is any set, the identity morphism assigned to A is given by (A, id_A, A) where $id_A \subseteq A \times A$ is the identity graph. We leave the reader to check as an *Exercise* that composition is an associative operation and that composition by identities is unitary. *Note: we now follow informal practice, and talk about "functions" f as morphisms, even though f is strictly speaking the graph component of the function* (A, f, B).
2. The category of sets and partial functions, $\mathcal{P}art$. The objects are sets and the morphisms are partial functions. The definition of composition is the expected one, namely given $f\colon A \to B$, $g\colon B \to C$, then for each element a of

A, $g \circ f(a)$ is defined with value $g(f(a))$ if both $f(a)$ and $g(f(a))$ are defined, and is otherwise not defined.

3. Any preordered set $(X, \leq)$ may be viewed as a category. Recall that a preorder $\leq$ on a set X is a reflexive, transitive relation on X. The set of objects is X. The set of morphisms is $\leq$. $(X, \leq)$ forms a category with identity morphisms (x, x) for each object x (because $\leq$ is reflexive) and composition $(y, z) \circ (x, y) \stackrel{\text{def}}{=} (x, z)$ (because $\leq$ is transitive). Note for x and y elements of X, there is at most one morphism from x to y according to whether $x \leq y$ or not.
4. The category $\mathcal{P}reset$ has objects all preordered sets, and morphisms the **monotone** functions. More precisely, a morphism with source $(X, \leq_X)$ and target $(Y, \leq_Y)$ is specified by giving a function $f: X \to Y$ such that if $x \leq_X x'$ then $f(x) \leq_Y f(x')$.
5. A **discrete** category is one for which the only morphisms are identities. So a very simple example of a discrete category is given by regarding any set as a category in which the objects are the elements of the set, there is an identity morphism for each element, and there are no other morphisms.
6. The objects of the category $\mathbb{F}$ are the elements $n \in \mathbb{N}$, where we regard n as the set $\{0, \ldots, n-1\}$ for $n \geq 1$, and 0 is the empty set $\varnothing$. A morphism $\rho: n \to n'$ is any set-theoretic function.
7. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. The **product category** $\mathcal{C} \times \mathcal{D}$ has as objects all pairs (C, D) where C is an object of $\mathcal{C}$ and D is an object of $\mathcal{D}$, and the morphisms are the obvious pairs $(f, g): (C, D) \to (C', D')$.

It is an *Exercise* to work through the details of these examples.

In category theory, a notion which is pervasive is that of *isomorphism.* If two objects are isomorphic, then they are very similar, but not necessarily identical. In the case of $\mathcal{S}et$, two sets X and Y are isomorphic just in case there is a bijection between them—informally, they have the same number of elements. The usual definition of bijection is that there is a function $f: X \to Y$ which is injective and surjective. Equivalently, there are functions $f: X \to Y$ and $g: Y \to X$ which are mutually inverse. We can use the idea that a pair of mutually inverse functions in the category $\mathcal{S}et$ gives rise to bijective sets to define the notion of isomorphism in an arbitrary category.

A morphism $f: A \to B$ in a category $\mathcal{C}$ is said to be an **isomorphism** if there is some $g: B \to A$ for which $f \circ g = id_B$ and $g \circ f = id_A$. We say that g is an **inverse** for f and that f is an inverse for g. Given objects A and B in $\mathcal{C}$, we say that A is **isomorphic** to B and write $A \cong B$ if such a mutually inverse pair of morphisms exists, and we say that the pair of morphisms **witnesses** the fact that $A \cong B$. Note that there may be many such pairs. In the category determined by a partially ordered set, the only isomorphisms are the identities, and in a preorder X with $x, y \in X$ we have $x \cong y$ iff $x \leq y$ and $y \leq x$. Note that in this case there can be only one pair of mutually inverse morphisms witnessing the fact that $x \cong y$. Here are a couple of *Exercises*.

(1) Let $\mathcal{C}$ be a category and let $f: A \to B$ and $g, h: B \to A$ be morphisms. If $f \circ h = id_B$ and $g \circ f = id_A$ show that $g = h$. Deduce that any morphism f has a **unique** inverse if such exists.
(2) Let $\mathcal{C}$ be a category and $f: A \to B$ and $g: B \to C$ be morphisms. If f and g are isomorphisms, show that $g \circ f$ is too. What is its inverse?

3.2 Functors

A function $f: X \to Y$ is a relation between two sets. We can think of the function f as specifying an element of Y for each element of X; from this point of view, f is rather like a program which outputs a value $f(x) \in Y$ for each $x \in X$. We might say that the element $f(x)$ is **assigned** to x. A functor is rather like a function between two categories. Roughly, a functor from a category $\mathcal{C}$ to a category $\mathcal{D}$ is an assignment which sends each object of $\mathcal{C}$ to an object of $\mathcal{D}$, and each morphism of $\mathcal{C}$ to a morphism of $\mathcal{D}$. This assignment has to satisfy some rules. For example, the identity on an object A of $\mathcal{C}$ is sent to the identity in $\mathcal{D}$ on the object FA, where the functor sends the object A in $\mathcal{C}$ to FA in $\mathcal{D}$. Further, if two morphisms in $\mathcal{C}$ compose, then their images under the functor must compose in $\mathcal{D}$. Very informally, we might think of the functor as "preserving the structure" of $\mathcal{C}$. Let us move to the formal definition of a functor.

A **functor** F between categories $\mathcal{C}$ and $\mathcal{D}$, written $F: \mathcal{C} \to \mathcal{D}$, is specified by

- an operation assigning objects FA in $\mathcal{D}$ to objects A in $\mathcal{C}$, and
- an operation assigning morphisms $Ff: FA \to FB$ in $\mathcal{D}$, to morphisms $f: A \to B$ in $\mathcal{C}$,

for which $F(id_A) = id_{FA}$, and whenever the composition of morphisms $g \circ f$ is defined in $\mathcal{C}$ we have $F(g \circ f) = Fg \circ Ff$. Note that $Fg \circ Ff$ is defined in $\mathcal{D}$ whenever $g \circ f$ is defined in $\mathcal{C}$, that is, Ff and Fg are composable in $\mathcal{D}$ whenever f and g are composable in $\mathcal{C}$.

Sometimes we give the specification of a functor F by writing the operation on an object A as $A \mapsto FA$ and the operation on a morphism f, where $f: A \to B$, as $f: A \to B \mapsto Ff: FA \to FB$. Provided that everything is clear, we sometimes say "the functor $f: \mathcal{C} \to \mathcal{D}$ is defined by an assignment

$$f: A \longrightarrow B \quad \mapsto \quad Ff: FA \longrightarrow FB$$

where $f: A \to B$ is any morphism of $\mathcal{C}$." We refer informally to $\mathcal{C}$ as the source of the functor F, and to $\mathcal{D}$ as the target of F. Here are some examples.

1. Let $\mathcal{C}$ be a category. The **identity** functor $id_{\mathcal{C}}$ is defined by $id_{\mathcal{C}}(A) \stackrel{\text{def}}{=} A$ where A is any object of $\mathcal{C}$ and $id_{\mathcal{C}}(f) \stackrel{\text{def}}{=} f$ where f is any morphism of $\mathcal{C}$.
2. We may define a functor $F: \mathcal{S}et \to \mathcal{S}et$ by taking the operation on objects to be $FA \stackrel{\text{def}}{=} [A]$ and the operation on morphisms $Ff \stackrel{\text{def}}{=} map(f)$, where the function $map(f): [A] \to [B]$ is defined by

$$map(f)(as) \stackrel{\text{def}}{=} \text{case of} \begin{cases} \epsilon \to \epsilon \\ ([a_0, \ldots, a_{l-1}]) = [f(a_0), \ldots, f(a_{l-1})] \end{cases}$$

Being our first example, we give explicit details of the verification that F is indeed a functor. To see that $F(id_A) = id_{FA}$ note that on non-empty lists

$$\begin{aligned} F(id_A)([a_0, \ldots, a_{l-1}]) &\overset{\text{def}}{=} map(id_A)([a_0, \ldots, a_{l-1}]) \\ &= [a_0, \ldots, a_{l-1}] \\ &= id_{[A]}([a_0, \ldots, a_{l-1}]) \\ &\overset{\text{def}}{=} id_{FA}([a_0, \ldots, a_{l-1}]), \end{aligned}$$

and to see that $F(g \circ f) = Fg \circ Ff$ note that

$$\begin{aligned} F(g \circ f)([a_0, \ldots, a_{l-1}]) &\overset{\text{def}}{=} map(g \circ f)([a_0, \ldots, a_{l-1}]) \\ &= [g(f(a_0)), \ldots, g(f(a_{l-1}))] \\ &= map(g)([f(a_0), \ldots, f(a_{l-1})]) \\ &= map(g)(map(f)([a_0, \ldots, a_{l-1}])) \\ &= Fg \circ Ff([a_0, \ldots, a_{l-1}]). \end{aligned}$$

3. Given categories $\mathcal{C}$ and $\mathcal{D}$ and an object D of $\mathcal{D}$, the **constant** functor $\tilde{D}: \mathcal{C} \to \mathcal{D}$ sends any object A of $\mathcal{C}$ to D and any morphism $f: A \to B$ of $\mathcal{C}$ to $id_D: D \to D$.
4. Given a set A, recall that the powerset $\mathcal{P}(A)$ is the set of subsets of A. We can define a functor $\mathcal{P}(f): \mathcal{S}et \to \mathcal{S}et$ which is given by

$$f: A \to B \quad \mapsto \quad \mathcal{P}(f): \mathcal{P}(A) \to \mathcal{P}(B),$$

where $f: A \to B$ is a function and $\mathcal{P}(f)$ is defined by

$$\mathcal{P}(f)(A') \overset{\text{def}}{=} \{f(a') \mid a' \in A'\}$$

where $A' \in \mathcal{P}(A)$. We call $\mathcal{P}(f): \mathcal{S}et \to \mathcal{S}et$ the **covariant powerset** functor.
5. Given functors $F: \mathcal{C} \to \mathcal{C}'$ and $G: \mathcal{D} \to \mathcal{D}'$, the **product functor**

$$F \times G: \mathcal{C} \times \mathcal{D} \to \mathcal{C}' \times \mathcal{D}'$$

assigns the morphism $(Ff, Gg): (FC, FD) \to (FC', FD')$ to any morphism $(f, g): (C, D) \to (C', D')$ in $\mathcal{C} \times \mathcal{D}$.
6. Any functor between two preorders A and B regarded as categories is precisely a monotone function from A to B.

It is an *Exercise* to check that the definitions define functors between categories.

3.3 Natural Transformations

Let $\mathcal{C}$ and $\mathcal{D}$ be categories and $F, G: \mathcal{C} \to \mathcal{D}$ be functors. Then a **natural transformation** α from F to G, written $\alpha: F \to G$, is specified by an operation which assigns to each object A in $\mathcal{C}$ a morphism $\alpha_A : FA \to GA$ in $\mathcal{D}$, such that for any morphism $f : A \to B$ in $\mathcal{C}$, we have $Gf \circ \alpha_A = \alpha_B \circ Ff$. We denote this equality by the following diagram

$$\begin{tikzcd}
FA \arrow[r, "\alpha_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\alpha_B"'] & GB
\end{tikzcd}$$

which is said to **commute**. The morphism α_A is called the **component** of the natural transformation α at A. We also write $\alpha : F \to G : \mathcal{C} \to \mathcal{D}$ to indicate that α is a natural transformation between the functors $F, G : \mathcal{C} \to \mathcal{D}$. If we are given such a natural transformation, we refer to the above commutative square by saying "consider naturality of α at $f : A \to B$."

1. Recall the functor $F : \mathcal{S}et \to \mathcal{S}et$ defined on page 143. We can define a natural transformation $rev : F \to F$ which has components $rev_A : [\,A\,] \to [\,A\,]$ defined by
$$rev_A(as) \overset{\text{def}}{=} \text{case of} \begin{cases} \epsilon \to \epsilon \\ [\,a_0, \ldots, a_{l-1}\,] \to [a_{l-1}, \ldots, a_0] \end{cases}$$
where $[\,A\,]$ is the set of finite lists over A (see Appendix). It is trivial to see that this does define a natural transformation:
$$Ff \circ rev_A([a_0, \ldots, a_{l-1}]) = [f(a_{l-1}), \ldots, f(a_0)] = rev_B \circ Ff([a_0, \ldots, a_{l-1}]).$$

2. Let X and A be sets. Write $X \to A$ for the set of functions from X to A, and let $(X \to A) \times X$ be the usual cartesian product of sets. Define a functor $F_X : \mathcal{S}et \to \mathcal{S}et$ by setting $F_X(A) \overset{\text{def}}{=} (X \to A) \times X$ where A is any set and letting
$$F_X(f) : (X \to A) \times X \longrightarrow (X \to B) \times X$$
be the function defined by $(g, x) \mapsto (f \circ g, x)$ where $f : A \to B$ is any function and $(g, x) \in (X \to A) \times X$. Then we can define a natural transformation $ev : F_X \to id_{\mathcal{S}et}$ by setting $ev_A(g, x) \overset{\text{def}}{=} g(x)$. To see that we have defined a natural transformation ev with components $ev_A : (X \to A) \times X \to A$ let $f : A \to B$ be a set function, $(g, x) \in (X \to A) \times X$, and note that
$$\begin{aligned}
(id_{\mathcal{S}et}(f) \circ ev_A)(g, x) &= f(ev_A(g, x)) \\
&= f(g(x)) \\
&= ev_B(f \circ g, x) \\
&= ev_B(F_X(f)(g, x)) \\
&= (ev_B \circ F_X(f))(g, x).
\end{aligned}$$

It will be convenient to introduce some notation for dealing with methods of "composing" functors and natural transformations. Let $\mathcal{C}$ and $\mathcal{D}$ be categories and let F, G, H be functors from $\mathcal{C}$ to $\mathcal{D}$. Also let $\alpha: F \to G$ and $\beta: G \to H$ be natural transformations. We can define a natural transformation $\beta \circ \alpha: F \to H$ by setting the components to be $(\beta \circ \alpha)_A \stackrel{\text{def}}{=} \beta_A \circ \alpha_A$. This yields a category $\mathcal{D}^{\mathcal{C}}$ with objects functors from $\mathcal{C}$ to $\mathcal{D}$, morphisms natural transformations between such functors, and composition as given above. $\mathcal{D}^{\mathcal{C}}$ is called the **functor category** of $\mathcal{C}$ and $\mathcal{D}$. As an *Exercise*, given categories $\mathcal{C}$ and $\mathcal{D}$, verify that $\mathcal{D}^{\mathcal{C}}$ is indeed a category. For example, one thing to check is that $\beta \circ \alpha$ as defined above is indeed natural.

An isomorphism in a functor category is referred to as a **natural isomorphism**. If there is a natural isomorphism between the functors F and G, then we say that F and G are **naturally isomorphic**.

Lemma 1. *Let $\alpha: F \to G: \mathcal{C} \to \mathcal{D}$ be a natural transformation. Then α is a natural isomorphism just in case each component α_C is an isomorphism in $\mathcal{D}$. More precisely, if we are given a natural isomorphism α in $\mathcal{D}^{\mathcal{C}}$ with inverse β, then each β_C is an inverse for α_C in $\mathcal{D}$; and if given a natural transformation α in $\mathcal{D}^{\mathcal{C}}$ for which each component α_C has an inverse (say β_C) in $\mathcal{D}$, then the β_C are the components of a natural transformation β which is the inverse of α in $\mathcal{D}^{\mathcal{C}}$.*

Proof. Direct calculations from the definitions, left as an *Exercise.*

3.4 Products

The notion of "product of two objects in a category" can be viewed as an abstraction of the idea of a cartesian product of two sets. The definition of a cartesian product of sets is an "internal" one; we specify the elements of the product in terms of the elements of the sets from which the product is composed. However the cartesian product has a particular property, namely

(*Property Φ*) *Given* any two sets A and B, *then* there is a set P and functions $\pi: P \to A$, $\pi': P \to B$ *such that* the following condition holds: given any functions $f: C \to A$, $g: C \to B$ with C any set, then there is a unique function $h: C \to P$ making the diagram

$$\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, dashed, "h"] \arrow[dr, "g"] & \\
A & P \arrow[l, "\pi"] \arrow[r, "\pi'"'] & B
\end{tikzcd}$$

commute. End of definition of (*Property Φ*).

Let us investigate an instance of (*Property Φ*) in the case of two given sets A and B. Suppose that $A \stackrel{\text{def}}{=} \{a, b\}$ and $B \stackrel{\text{def}}{=} \{c, d, e\}$. Let us take P to be $A \times B \stackrel{\text{def}}{=} \{(x, y) \mid x \in A, y \in B\}$ and the functions π and π' to be coordinate projection to A and B respectively, and see if (P, π, π') makes the instance of (*Property Φ*) for the given A and B hold. Let C be any other set and $f\colon C \to A$ and $g\colon C \to B$ be any two functions. Define the function $h\colon C \to P$ by $z \mapsto (f(z), g(z))$. We leave the reader to verify that indeed $f = \pi \circ h$ and $g = \pi' \circ h$, and that h is the only function for which these equations hold with the given f and g. Now define $P' \stackrel{\text{def}}{=} \{1,2,3,4,5,6\}$ along with functions $p\colon P' \to A$ and $q\colon P' \to B$ where

$$\begin{array}{llll} p(1), & p(2), & p(3) = a & \qquad q(1), \quad q(4) = c \\ p(4), & p(5), & p(6) = b & \qquad q(2), \quad q(5) = d \\ & & & \qquad q(3), \quad q(6) = e \end{array}$$

In fact (P', p, q) also makes the instance of (*Property* Φ) for the given A and B hold true. To see this, one can check by enumerating six cases that there is a unique function $h\colon C \to P'$ for which $f = p \circ h$ and $g = q \circ h$ (for example, if $x \in C$ and $f(x) = a$ and $g(x) = d$ then we must have $h(x) = 2$, and this is one case).

Now notice that there is a bijection between P (the cartesian product

$$\{(a,c),(a,d),(a,e),(b,c),(b,d),(b,e)\}$$

of A and B) and P'. In fact any choices for the set P can be shown to be bijective. It is very often useful to determine sets up to bijection rather than worry about their elements or "internal make up," so we might consider taking (*Property* Φ) as a definition of cartesian product of two sets and think of the P and P' in the example above as two implementations of the notion of cartesian product of the sets A and B. Of course (*Property* Φ) only makes sense when talking about the collection of sets and functions; we can give a definition of cartesian product for an arbitrary category which is exactly (*Property* Φ) for the "category" of sets and functions.

A **binary product** of objects A and B in a category $\mathcal{C}$ is specified by

- an object $A \times B$ of $\mathcal{C}$, together with
- two **projection** morphisms $\pi_A\colon A \times B \to A$ and $\pi_B\colon A \times B \to B$,

for which given any object C and morphisms $f\colon C \to A$, $g\colon C \to B$, there is a unique morphism $\langle f, g\rangle\colon C \to A \times B$ for which $\pi_A \circ \langle f, g\rangle = f$ and $\pi_B \circ \langle f, g\rangle = g$.

We refer simply to a binary product $A \times B$ instead of $(A \times B, \pi_A, \pi_B)$, without explicit mention of the projection morphisms. The data for a binary product is more readily understood as a commutative diagram, where we have written $\exists!$ to mean "there exists a unique":

$$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, dashed, "\exists!\ \langle f, g\rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\pi_A"] \arrow[r, "\pi_B"'] & B \end{tikzcd}$$

Given a binary product $A \times B$ and morphisms $f\colon C \to A$ and $g\colon C \to B$, the unique morphism $\langle f, g\rangle\colon C \to A \times B$ (making the above diagram commute) is called the **mediating** morphism for f and g. We sometimes refer to a property which involves the "existence of a unique morphism" leading to a structure which is determined up to isomorphism as a **universal** property. We also call $\langle f, g\rangle$ the **pair** of f and g. We say that the category $\mathcal{C}$ **has binary products** if there is a product in $\mathcal{C}$ of any two objects A and B, and that $\mathcal{C}$ has **specified** binary products if there is a given canonical choice of binary product for each pair of objects. For example, in $\mathcal{S}et$ we can specify binary products by setting $A \times B \stackrel{\text{def}}{=} \{\ (a,b) \ \mid \ a \in A, b \in B\ \}$ with projections given by the usual set-theoretic projection functions. Talking of **specified** binary products is a reasonable thing to do: given A and B, any binary product of A and B will be isomorphic to the specified $A \times B$.

Lemma 2. *A binary product of A and B in a category $\mathcal{C}$ is unique up to isomorphism if it exists.*

Proof. Suppose that (P, p_A, p_B) and (P', p'_A, p'_B) are two candidates for the binary product. Then we have $\langle p_A, p_B\rangle: P \to P'$ by applying the defining property of (P', p'_A, p'_B) to the morphisms $p_A: P \to A$, $p_B: P \to B$, and further $\langle p'_A, p'_B\rangle: P' \to P$ exists from a similar argument. So we have diagrams of the form

$$\begin{tikzcd}
 & P \arrow[dl, "p_A"'] \arrow[d, dashed, "\langle p_A, p_B\rangle"] \arrow[dr, "p_B"] & \\
A & P' \arrow[l, "p'_A"] \arrow[r, "p'_B"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
 & P' \arrow[dl, "p'_A"'] \arrow[d, dashed, "\langle p'_A, p'_B\rangle"] \arrow[dr, "p'_B"] & \\
A & P \arrow[l, "p_A"] \arrow[r, "p_B"'] & B
\end{tikzcd}$$

But then $f \stackrel{\text{def}}{=} \langle p'_A, p'_B\rangle \circ \langle p_A, p_B\rangle: P \to P$ and one can check that $p_A \circ f = p_A$ and that $p_B \circ f = p_B$, that is f is a mediating morphism for the binary product (P, p_A, p_B); we can picture this as the following commutative diagram:

$$\begin{tikzcd}
 & P \arrow[dl, "p_A"'] \arrow[d, dashed, "f"] \arrow[dr, "p_B"] & \\
A & P \arrow[l, "p_A"] \arrow[r, "p_B"'] & B
\end{tikzcd}$$

But it is trivial that id_P is also such a mediating morphism, and so uniqueness implies $f = id_P$. Similarly one proves that $\langle p_A, p_B\rangle \circ \langle p'_A, p'_B\rangle = id_{P'}$, to deduce $P \cong P'$ witnessed by the morphisms $\langle p_A, p_B\rangle$ and $\langle p'_A, p'_B\rangle$.

Here are some examples

1. The category $\mathcal{P}reset$ has binary products. Given objects $A \stackrel{\text{def}}{=} (X, \leq_X)$ and $B \stackrel{\text{def}}{=} (Y, \leq_Y)$, the product object $A \times B$ is given by $(X \times Y, \leq_{X \times Y})$ where $X \times Y$ is cartesian product, and $(x, y) \leq_{X \times Y} (x', y')$ just in case $x \leq_X x'$ and $y \leq_Y y'$. It is an *Exercise* to check the details.
2. The category $\mathcal{P}art$ has binary products. Given objects A and B, the binary product object is defined by

   $$(A \times B) \cup (A \times \{ *_A \}) \cup (B \times \{ *_B \})$$

   where $\times$ is cartesian product, and $*_A$ and $*_B$ are distinct elements not in A nor in B. The project π_A is undefined on $B \times \{ *_B \}$ and π_B is undefined on $A \times \{ *_A \}$. Of course $\pi_A(a, *_A) = a$ for all $a \in A$, and $\pi_B(b, *_B) = b$ for all $b \in B$.
3. The category $\mathbb{F}$ has binary products. The product of n and m is written $n \times m$ and is given by $n * m$, that is, the set $\{ 0, \ldots, (n * m) - 1 \}$. We leave it as an *Exercise* to formulate possible definitions of projection functions. *Hint: Think about the general illustration of products at the start of this section.*
4. Consider a preorder $(X, \leq)$ which has all binary meets $x \wedge y$ for $x, y \in X$ as a category. It is an *Exercise* to verify that binary meets are binary products.

It will be useful to have a little additional notation which will be put to use later on.

We can define the **ternary product** $A \times B \times C$ for which there are three projections, and any mediating morphism can be written $\langle f, g, h \rangle$ for suitable f, g and h. It is an *Exercise* to make all the definitions and details precise.

Let $\mathcal{C}$ be a category with finite products and take morphisms $f\colon A \to B$ and $f'\colon A' \to B'$. We write

$$f \times f'\colon A \times A' \to B \times B'$$

for the morphism $\langle f \circ \pi, f' \circ \pi' \rangle$ where $\pi\colon A \times A' \to A$ and $\pi'\colon A \times A' \to A'$. The uniqueness condition (universal property) of mediating morphisms means that in general one has

$$id_A \times id_{A'} = id_{A \times A'} \qquad \text{and} \qquad (g \times g') \circ (f \times f') = g \circ f \times g' \circ f',$$

where $g\colon B \to C$ and $g'\colon B' \to C'$. Thus in fact we have a functor

$$\times\colon \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$$

where $\mathcal{C} \times \mathcal{C}$ is a product of categories. Note that we sometimes write the image of (A, A) or (f, f) under $\times$ as A^2 or f^2.

1. Verify the equalities (uniqueness conditions) given above.
2. Let $\mathcal{C}$ be a category with finite products and let

   $$\begin{array}{lll} l\colon X \to A & f\colon A \to B & g\colon A \to C \\ h\colon B \to D & k\colon C \to E & \end{array}$$

   be morphisms of $\mathcal{C}$. Show that

   $$(h \times k) \circ \langle f, g \rangle = \langle h \circ f, k \circ g \rangle \qquad \langle f, g \rangle \circ l = \langle f \circ l, g \circ l \rangle$$

3.5 Coproducts

A coproduct is a dual notion of product. A **binary coproduct** of objects A and B in a category $\mathcal{C}$ is specified by

- an object $A+B$ of $\mathcal{C}$, together with
- two **insertion** morphisms $\iota_A\colon A \to A+B$ and $\iota_B\colon B \to A+B$,

such that for each pair of morphisms $f\colon A \to C$, $g\colon B \to C$ there exists a unique morphism $[f,g]\colon A+B \to C$ for which $[f,g] \circ \iota_A = f$ and $[f,g] \circ \iota_B = g$. We can picture this definition through the following commutative diagram:

$$\begin{tikzcd}
A \arrow[r, "\iota_A"] \arrow[dr, "f"'] & A+B \arrow[d, dashed, "{[f,g]}"] & B \arrow[l, "\iota_B"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

1. In the category $\mathcal{S}et$, the binary coproduct of sets A and B is given by their disjoint union together with the obvious insertion functions. We can define the disjoint union $A+B$ of A and B as the union $(A \times \{1\}) \cup (B \times \{2\})$ with the insertion functions
$$\iota_A : A \to A+B \leftarrow B : \iota_B$$
where ι_A is defined by $a \mapsto (a,1)$ for all $a \in A$, and ι_B is defined analogously. Given functions $f\colon A \to C$ and $g\colon B \to C$, then $[f,g]\colon A+B \to C$ is defined by
$$[f,g](\xi) \stackrel{\text{def}}{=} \mathsf{case}\ \xi\ \mathsf{of}$$
$$\iota_A(\xi_A) = (\xi_A,1) \mapsto f(\xi_A)$$
$$\iota_B(\xi_B) = (\xi_B,2) \mapsto f(\xi_B)$$
We sometimes say that $[f,g]$ is defined by **case analysis**.
2. The category $\mathcal{P}reset$ has binary coproducts. Given objects $A \stackrel{\text{def}}{=} (X, \leq_X)$ and $B \stackrel{\text{def}}{=} (Y, \leq_Y)$, the product object $A+B$ is given by $(X+Y, \leq_{X+Y})$ where $X+Y$ is disjoint union, and $\xi \leq_{X+Y} \xi'$ just in case $\xi = (x,1)$ and $\xi' = (x',1)$ for some x, x' such that $x \leq_X x'$, or $\xi = (y,2)$ and $\xi' = (y',2)$ for some y, y' such that $y \leq_Y y'$. It is an *Exercise* to check the details.
3. In $\mathbb{F}$ the coproduct object of n and m is $n+m$ where we interpret $+$ as addition on $\mathbb{N}$. It is an *Exercise* to work out choices of coproduct insertions. What might we take as the *canonical* projections?
4. When does a preordered set have binary coproducts?

We end this section with notation that will be used later on.

One can define the **ternary coproduct** of three objects, which will have three insertions, and mediating morphisms $[f,g,h]$. Fill in the details as an *Exercise.* Of course we can generalize to an n-ary coproduct, and call the mediating morphism a **cotupling**.

Let $\mathcal{C}$ be a category with finite coproducts and take morphisms $f: A \to B$ and $f': A' \to B'$. We write

$$f + f': A + A' \to B + B'$$

for the morphism $[\iota_B \circ f, \iota_{B'} \circ f']$ where $\iota_B: B \to B + B'$ and $\iota_{B'}: B' \to B + B'$. The uniqueness condition of mediating morphisms means that one has

$$id_A + id_{A'} = id_{A+A'} \qquad \text{and} \qquad (g + g') \circ (f + f') = g \circ f + g' \circ f',$$

where $g: B \to C$ and $g': B' \to C'$. So we also have a functor

$$+: \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$$

In a category with binary coproducts, then for any morphisms f, g, h, k, l with certain source and target, the equalities

$$l \circ [f, g] = [l \circ f, l \circ g] \qquad\qquad [f, g] \circ (h + k) = (f \circ h) + (g \circ k)$$

always hold due to the universal property of (binary) coproducts. What are the sources and targets? Prove the equalities.

3.6 Algebras

Let F be an **endofunctor** on $\mathcal{C}$, that is a functor $F: \mathcal{C} \to \mathcal{C}$. An **algebra** for the functor F is specified by a pair (A, σ_A) where A is an object of $\mathcal{C}$ and $\sigma_A: FA \to A$ is a morphism. An **initial** F-algebra (I, σ_I) is an algebra for which given any other (A, σ_A), there is a unique morphism $\overline{f}: I \to A$ such that

$$\begin{tikzcd}
FI \arrow[r, "\sigma_I"] \arrow[d, "F\overline{f}"'] & I \arrow[d, "\overline{f}"] \\
FA \arrow[r, "\sigma_A"'] & A
\end{tikzcd}$$

commutes. As an *Exercise*, show that if $\sigma_I: FI \to I$ is initial, then so too is $\sigma_{I'}: FI' \to I'$ where I' is any object isomorphic to I, and $\sigma_{I'}$ is a morphism for you to define.

One reason for defining initial algebras is that certain datatypes can be modeled as instances. Here is the rough idea. Each constructor of a datatype can be thought of as a coproduct insertion[3]. Each constructor is applied to a finite number (eg 2) of expressions, constituting a tuple of expressions. This can be thought of as a product (eg binary). The datatype

$$\mathit{Exp} ::= \mathsf{V}\ \mathbb{V} \mid \mathsf{S}\ \mathit{Exp} \mid \mathsf{A}\ \mathit{Exp}\ \mathit{Exp}$$

[3] Such insertions must be injective. There are categories in which insertions are *not* injective! In all of the examples in these notes, however, insertions will be injective.

corresponds to (is modeled by) an object $V + E + (E \times E)$. The recursiveness of the datatype declaration is modeled by requiring

$$E \cong V + E + (E \times E) \qquad \dagger$$

In these notes we shall see how to solve an equation such as † in the category $\mathcal{S}et$. In fact if we define a functor $\Sigma\colon \mathcal{S}et \to \mathcal{S}et$ by $\Sigma\xi \overset{\text{def}}{=} \tilde{\mathbb{V}} + \xi + (\xi \times \xi)$, then it will turn out that the solution we would construct using the methods below is an initial algebra (σ_E, E). We now give a few examples, and illustrate the solution method.

3.7 The Functor $1 + (-)\colon \mathcal{S}et \longrightarrow \mathcal{S}et$

We write $1\colon \mathcal{S}et \to \mathcal{S}et$ for the functor which maps any function $f\colon A \to B$ to $id_{\{*\}}\colon \{*\} \to \{*\}$ where $\{*\}$ is a one element set. Note that we will often also write 1 for such a set. The functor maps $f\colon A \to B$ to $id_1 + f\colon 1 + A \to 1 + B$.

The initial algebra is $\mathbb{N}$ up to isomorphism. We show how to construct an initial algebra—the method will be applied in later sections, in adapted form, to produce models of datatypes which represent syntax (see Section 2).

We set $S_0 \overset{\text{def}}{=} \varnothing$ and $S_{r+1} \overset{\text{def}}{=} 1 + S_r$. Note that there is a coproduct insertion $\iota_S : S_r \to S_{r+1}$. Note also that there is an inclusion[4] function (morphism) $i_r\colon S_r \hookrightarrow S_{r+1}$ where $i_0 \overset{\text{def}}{=} \varnothing\colon S_0 \to S_1$, and $i_{r+1} \overset{\text{def}}{=} id_1 + i_r$. The difference is an elementary, but subtle, point! For example, we have $i_1, \iota_{S_1}\colon S_1 \to S_2 = 1 + S_1$ for which $\iota_{S_1}(*, 1) = ((*, 1), 2)$, where $(*, 1) \in S_1 = 1 + \varnothing$. But $i_1 = id_1 + i_0$ and so

$$i_1(*, 1) = [\iota_1 \circ id_1, \iota_{S_1} \circ i_0](*, 1) = \iota_1 \circ id_1(*) = (*, 1)$$

We also write $i'_r\colon S_r \hookrightarrow T$ where $T \overset{\text{def}}{=} \cup_r S_r$, and claim that T is the object part of an initial algebra for $1 + (-)$. Note that as $\sigma_T\colon 1 + T \to T$, σ_T must be the copair of two morphisms. We set $\sigma_T \overset{\text{def}}{=} [k, k']$ where $k\colon 1 \to T$ and $k'\colon T \to T$, with k and k' defined as follows. Note there is a function

$$\begin{tikzcd} 1 \arrow[r, "\iota_1"] & 1 + \varnothing = S_1 \arrow[r, "i'_1"] & T \end{tikzcd}$$

and we set $k \overset{\text{def}}{=} i'_1 \circ \iota_1$. Note there are also functions

$$\begin{tikzcd} S_r \arrow[r, "\iota_S"] & 1 + S_r = S_{r+1} \arrow[r, "i'_{r+1}"] & T \end{tikzcd}$$

whose composition we call k'_r. It is an *Exercise* to check that $k'_{r+1} \circ i_r = k'_r$ by induction on r. Hence we can legitimately define the function $k'\colon T \to T$ by setting $k'(\xi) \overset{\text{def}}{=} k'_r(\xi)$ for any r such that $\xi \in S_r$.

[4] That is, $S_r \subset S_{r+1}$ for all $r \geq 0$.

We have to verify that $\sigma_T: 1 + T \to T$ is an initial algebra, namely, there is exactly one commutative diagram

$$\begin{tikzcd}
1 + T \arrow[r, "\sigma_T"] \arrow[d, "id_1 + \overline{f}"'] & T \arrow[d, "\overline{f}"] \\
1 + A \arrow[r, "f"'] & A
\end{tikzcd}$$

for any such given f. We define a family of functions $\overline{f}_r: S_r \to A$ by setting $\overline{f}_0 \stackrel{\text{def}}{=} \varnothing: S_0 \to A$, and recursively $\overline{f}_{r+1} \stackrel{\text{def}}{=} [f \circ \iota_1, f \circ \iota_A \circ \overline{f}_r]$. It is an *Exercise* to check that $\overline{f}_{r+1} \circ i_r = \overline{f}_r$. Hence we can legitimately define $\overline{f}: T \to A$ by $\overline{f}(\xi) \stackrel{\text{def}}{=} \overline{f}_r(\xi)$ for any r where $\xi \in S_r$.

To check that the diagram commutes, we have to prove that

$$\overline{f} \circ [k, k'] = f \circ (id_1 + \overline{f})$$

By the universal property of coproducts, this is equivalent to showing

$$[\overline{f} \circ k, \overline{f} \circ k'] = [f \circ \iota_1, f \circ \iota_A \circ \overline{f}]$$

which we can do by checking that the respective components are equal. We give details for $\overline{f} \circ k' = f \circ \iota_A \circ \overline{f}$. Take any element $\xi \in T$. Then we have

$$\begin{aligned}
\overline{f}(k'(\xi)) &= \overline{f}(\iota_S\ (\xi)) \\
&= \overline{f}_{r+1}(\iota_S\ (\xi)) \\
&= [f \circ \iota_1, f \circ \iota_A \circ \overline{f}_r](\iota_S\ (\xi)) \\
&= f(\iota_A(\overline{f}_r(\xi))) \\
&= f(\iota_A(\overline{f}(\xi)))
\end{aligned}$$

The first equality is by definition of k' and k'_r; the second by definition of $\overline{f}$; the third by definition of $\overline{f}_{r+1}$.

A final *Exercise* is to check that $T \cong N$.

3.8 The Functor $A + (-): \mathcal{S}et \longrightarrow \mathcal{S}et$

Let A be a set, $+$ denote coproduct. Then the functor $A + (-)$ has an initial algebra $(A \times \mathbb{N}, \sigma_{A\times\mathbb{N}})$ where $\sigma_{A\times\mathbb{N}}: A + (A \times \mathbb{N}) \to A \times \mathbb{N}$ is defined by

$$\begin{aligned}
\sigma_{A\times\mathbb{N}}(\xi) \stackrel{\text{def}}{=} \ & \mathsf{case}\ \xi\ \mathsf{of} \\
& \quad \iota_A(a) \mapsto (a, 0) \\
& \quad \iota_{A\times\mathbb{N}}(a, n) \mapsto (a, n+1)
\end{aligned}$$

where ι_A and $\iota_{A\times\mathbb{N}}$ are the left and right coproduct insertions, and $n \geq 0$.

Then given any function $f: A + S \to S$, we can define $\overline{f}$ where

$$\begin{tikzcd} A + (A \times \mathbb{N}) \arrow[r, "\sigma_{A\times\mathbb{N}}"] \arrow[d, "id_A + \overline{f}"'] & A \times \mathbb{N} \arrow[d, "\overline{f}"] \\ A + S \arrow[r, "f"'] & S \end{tikzcd}$$

by setting

$$\overline{f}(a, 0) \stackrel{\text{def}}{=} f(\iota_A(a))$$
$$\overline{f}(a, n+1) \stackrel{\text{def}}{=} f(\iota_{A\times\mathbb{N}}(\overline{f}(a, n)))$$

It is an *Exercise* to verify that this *does* yield an initial algebra. It is also an exercise to construct an initial algebra using the methodology of the previous section, and to show that the two algebras are isomorphic.

3.9 The Functor $1 + (A \times -): \mathcal{S}et \to \mathcal{S}et$

For $k \geq 1$ we define the set A^k to be the collection of functions $\{ 1, \ldots, k \} \to A$. We identify A with A^1; an element $a \in A$ is essentially a function $1 \to A$. If $a \in A$ and $l \in A^k$, then we define $al \in A^{k+1}$ by $al(1) \stackrel{\text{def}}{=} a$ and $al(r) \stackrel{\text{def}}{=} l(r-1)$ for $r \geq 2$.

The functor $1 + (A \times -)$ has an initial algebra (L, σ_L), where we set $L \stackrel{\text{def}}{=} \{ \mathsf{nil} \} \cup (\bigcup_{1 \leq k < \omega} A^k)$, and $\sigma_L: 1 + (A \times L) \to L$ is defined by

$$\sigma_L(\iota_1(*)) \stackrel{\text{def}}{=} \mathsf{nil}$$
$$\sigma_L(\iota_{A\times L}(a, nil)) \stackrel{\text{def}}{=} a$$
$$\sigma_L(\iota_{A\times L}(a, l)) \stackrel{\text{def}}{=} al$$

4 Models of Syntax

Recall that in Section 2 we defined the syntax terms of an object level programming language by making use of recursive datatypes. Note the phrase "making use of". Recall (pages 135 and 152) that we would like to have an ideal situation (IS) in which

– we could write down a recursive datatype, whose elements are precisely the terms which we are interested in; and
– we have a mathematical model of our syntax which is given as a solution to a recursive equation derived directly from the datatype.

Unfortunately, this is not so easily done when we are considering syntax involving binding. It can be done (as is well known, and illustrated in these notes) for simple syntax terms which involve *algebraic* constructors, such as the example in Section 2.1. We can come close to (IS) for binding syntax, but we can't achieve it exactly using the traditional techniques described here.

In Section 2.1 we wrote down a datatype for expressions e, and then restricted attention to expressions for which $\Gamma^n \vdash^{\overline{\mathsf{db}}} e$, that is, the variables occurring in e must come from $v^0, \dots, v^{n-1}$. This does not conform to standard practice. We would expect to deal with expressions *specified using* metavariables x which would denote *any* of the actual variables v^i. We could of course change the judgement $\Gamma^n \vdash^{\overline{\mathsf{db}}} e$ to $\Delta \vdash^{\overline{\mathsf{db}}} e$ (as in Section 2.3 and recall the exercise at the end of Section 2) where Δ is *any* finite environment of metavariables $x_0, \dots, x_{n-1}$ and any x_j denotes an actual variable. In this case, the expressions e such that $\Delta \vdash^{\overline{\mathsf{db}}} e$ *correspond exactly* with the expressions given by the datatype, and moreover are the terms of the object syntax. Further, we have a mathematical model described in the first half of Section 4.1, and we manage to achieve (IS).

The reason for describing the judgements $\Gamma^n \vdash^{\overline{\mathsf{db}}} e$ in this simple setting, is that they illustrate a methodology which we must apply if we are to get near to (IS) when dealing with binding syntax. In order to "avoid" the extra infrastructure of α-equivalence, and define the terms of the λ-calculus inductively (although not *exactly* as the elements of a datatype), we can restrict to environments Γ^n so that we can choose unique, explicit binding variables in abstractions. Further, the use of environments Γ^n with variables chosen systematically from a specified order will be used crucially when formulating the mathematical models in presheaf categories. Here is the rough idea of how we use presheaves over $\mathbb{F}$ to model expressions. A presheaf associates to every n the set of expressions whose (free) variables appear in Γ^n. And given any morphism $\rho\colon n \to n'$ in $\mathbb{F}$, the presheaf associates to ρ a function which, roughly, maps any expression e to

$$e\{v^{\rho(0)}, \dots, v^{\rho(n-1)}/v^0, \dots, v^{n-1}\}$$

Thus ρ renames the (free) variables in e.

In this section we give some mathematical models of each system of syntax from Section 2. In each case, we

Step 1. define an abstract endofunctor (over $\mathcal{F} \overset{\text{def}}{=} \mathcal{S}et^{\mathbb{F}}$—see below) which bears similarities to the datatype in question;
Step 2. construct an initial algebra T for the endofunctor;
Step 3. show that the datatype and system of syntax gives rise to a functor $\mathit{Exp}\colon \mathbb{F} \to \mathcal{S}et$, that is a presheaf in $\mathcal{F}$;
Step 4. complete the picture by showing that $T \cong \mathit{Exp}$ so that T forms an abstract mathematical model of the syntax.

In order to consider categorical models of syntax, we shall require some notation and facts concerning a particular functor category. Recall the category $\mathbb{F}$ defined on page 142. The functor category which will be of primary concern is $\mathcal{S}et^{\mathbb{F}}$. We will write $\mathcal{F}$ for it.

A typical object $F\colon \mathbb{F} \to \mathcal{S}et$ is an example[5] of a **presheaf** over $\mathbb{F}$. A simple example is given by the powerset functor $\mathcal{P}$, defined on page 144, restricted to sets of the form n. Another (trivial) example is the **empty presheaf** $\varnothing$ which maps any $\rho\colon n \to m$ to the empty function with empty source and target.

The category $\mathcal{F} \overset{\text{def}}{=} \mathcal{S}et^{\mathbb{F}}$ has binary products. If F and F' are objects in $\mathcal{F}$, the product object $F \times F'\colon \mathbb{F} \to \mathcal{S}et$ is defined on objects n in $\mathbb{F}$ by

$$(F \times F')n \overset{\text{def}}{=} (Fn) \times (F'n).$$

Now let $\rho\colon n \to n'$ be a morphism in $\mathbb{F}$. Hence $(F \times F')\rho$ should be a morphism with source and target

$$(Fn) \times (F'n) \longrightarrow (Fn') \times (F'n')$$

In fact we define $(F \times F')\rho \overset{\text{def}}{=} (F\rho) \times (F'\rho)$ where we make use of the functor $\times\colon \mathcal{S}et \times \mathcal{S}et \longrightarrow \mathcal{S}et$. The projection $\pi_F\colon F \times F' \to F$ is defined by giving components $(\pi_F)_n \overset{\text{def}}{=} \pi_{Fn}$ where n is an object of $\mathbb{F}$, with $\pi_{F'}$ defined similarly. Check the details as an *Exercise*.

The category also has binary coproducts. Let ξ be any object or morphism of $\mathbb{F}$. We define $F + F'$ by setting $(F + F')\xi \overset{\text{def}}{=} (F\xi) + (F'\xi)$. The insertion morphism (natural transformation) $\iota_F\colon F + F' \to F$ has components $(\iota_F)_n \overset{\text{def}}{=} \iota_{Fn}\colon (Fn) + (F'n) \to Fn$. ι'_F is defined analogously. We leave it as an *Exercise* to verify that $F + F'$ is indeed a functor, that the insertions are natural, and that the definitions above do give rise to binary coproducts.

Now let F and F' be two such objects (presheaves). Suppose that for any n in $\mathbb{F}$, $F'n \subset Fn$, and that the diagram

$$\begin{tikzcd}
F'n \arrow[r, phantom, "\subset"] \arrow[d, "F'\rho"'] & Fn \arrow[d, "F\rho"] \\
F'n' \arrow[r, phantom, "\subset"] & Fn'
\end{tikzcd}$$

commutes for any $\rho\colon n \to n'$. This gives rise to a natural transformation, which we denote by $i\colon F' \hookrightarrow F$.

We also need a functor $\delta\colon \mathcal{F} \to \mathcal{F}$. Suppose that F is an object in $\mathcal{F}$. Then $\delta\ F$ is defined by assigning to any morphism $\rho\colon n \to n'$ in $\mathbb{F}$ the function

$$(\delta\ F)\rho \overset{\text{def}}{=} F(\rho + id_1)\colon F(n+1) \longrightarrow F(n'+1)$$

If $\alpha\colon F \to F'$ in $\mathcal{F}$, then the components of $\delta\ \alpha$ are given by $(\delta\ \alpha)_n \overset{\text{def}}{=} \alpha_{n+1}$. It is an *Exercise* to verify all the details.

Lemma 3. *Suppose that $(S_r \mid r \geq 0)$ is a family of presheaves in $\mathcal{F}$, with $i_r\colon S_r \hookrightarrow S_{r+1}$ for each r. Then there is a* ***union*** *presheaf T in $\mathcal{F}$, such that $i'_r\colon S_r \hookrightarrow T$. We sometimes write $\cup_r S_r$ for T.*

[5] In general, we call $\mathcal{S}et^{\mathcal{C}}$ the category of presheaves over $\mathcal{C}$.

Proof. Let $\rho\colon n \to n'$ be any morphism in $\mathbb{F}$. Then we define $Tn \overset{\text{def}}{=} \bigcup_r S_r n$, and the function $T\rho\colon Tn \to Tn'$ is defined by $(T\rho)(\xi) \overset{\text{def}}{=} (S_r\rho)(\xi)$ where $\xi \in Tn$, and r is any index for which $\xi \in S_r(n)$. It is an *Exercise* to verify that we have defined a functor. Why is it well-defined? *Hint: prove that*

$$(\forall r, r' \geq 0)(r' \geq r \implies S_r \hookrightarrow S_{r'})$$

Lemma 4. *Let $(\phi_r\colon S_r \to A \mid r \geq 0)$ be a family of natural transformations in $\mathcal{F}$ with the S_r as in Lemma 3, and such that $\phi_{r+1} \circ i_r = \phi_r$. Then there is a unique natural transformation $\phi\colon T \to A$, such that $\phi \circ i'_r = \phi_r$.*

Proof. It is an *Exercise* to prove the lemma. The proof requires a simple calculation using the definitions. *Hint: Note that there are functions $\phi_n\colon Tn \to An$ where we set $\phi_n(\xi) \overset{\text{def}}{=} (\phi_r)_n(\xi)$ for $\xi \in S_r n$. The conditions of the lemma (trivially) imply the existence and uniqueness of the ϕ_n, which are natural in n.*

4.1 A Model of Syntax with Distinguished Variables and without Binding

Step 1. We define a functor Σ_{Var} which "corresponds" to the signature of Section 2.1. First, we define the functor $Var\colon \mathbb{F} \to \mathcal{S}et$. Let $\rho\colon m \to n$ in $\mathbb{F}$. Then we set $Var\, m \overset{\text{def}}{=} \{ v^0, \dots, v^{m-1} \}$ and $Var\, \rho(v^i) \overset{\text{def}}{=} v^{\rho i}$. It is trivial that Var is a functor. Recall that $\mathcal{S}et^{\mathbb{F}}$ has finite products and coproducts, and moreover that the operations $+$ and $\times$ can be regarded as functors. Thus we can define a functor $\Sigma_{Var}\colon \mathcal{S}et^{\mathbb{F}} \to \mathcal{S}et^{\mathbb{F}}$ by setting $\Sigma_{Var}\,\xi \overset{\text{def}}{=} Var + \xi + \xi^2$ where ξ is either an object or a morphism.

Step 2. We now show that the functor Σ_{Var} has an initial algebra, which we denote by $\sigma_T\colon \Sigma_{Var}\, T \to T$. We define T as the union of a family of presheaves $(S_r \mid r \geq 0)$ which satisfy the conditions of Lemma 3. We set $S_0 \overset{\text{def}}{=} \varnothing$ which is the empty presheaf, and then set

$$S_{r+1} \overset{\text{def}}{=} \Sigma_{Var}\, S_r = Var + S_r + S_r^2$$

We now check that the conditions of Lemma 3 hold, that is, $i_r\colon S_r \hookrightarrow S_{r+1}$ for all $r \geq 0$. We use induction over r. It is immediate that $i_0\colon S_0 \hookrightarrow S_1$ from the definition of S_0. Now suppose that for any r, $i_r\colon S_r \hookrightarrow S_{r+1}$. We are required to show that $i_{r+1}\colon S_{r+1} \hookrightarrow S_{r+2}$, that is, for any n in $\mathbb{F}$,

$$\begin{tikzcd}
Var\, n + S_r n + (S_r n)^2 \arrow[r, phantom, "\subset"] \arrow[d, "Var\,\rho + S_r\rho + (S_r\rho)^2"'] & Var\, n + S_{r+1} n + (S_{r+1} n)^2 \arrow[d, "Var\,\rho + S_{r+1}\rho + (S_{r+1}\rho)^2"] \\
Var\, n' + S_r n' + (S_r n')^2 \arrow[r, phantom, "\subset"] & Var\, n' + S_{r+1} n' + (S_{r+1} n')^2
\end{tikzcd}$$

It follows from the induction hypothesis, $S_r n \subset S_{r+1} n$, and the definition of $+$ and $\times$ in $\mathcal{S}et$, that we have subsets as indicated in the diagram. As an *Exercise* make sure that you understand this—you need to examine the definitions of $\times$ and $+$. In fact the top inclusion is the component at n of the natural transformation $\Sigma_{Var}\, i_r = id_{Var} + i_r + i_r^2$. Thus we have $i_{r+1} = \Sigma_{Var}\, i_r$. It is an *Exercise* to check that the diagram commutes. Thus we can define $T \overset{\text{def}}{=} \bigcup_r S_r$ in $\mathcal{F}$.

Next we consider the structure map σ_T. This natural transformation in $\mathcal{F}$ has source and target $Var + T + T^2 \to T$ and so it is given by $\sigma_T \overset{\text{def}}{=} [\kappa, \kappa', \kappa'']$, the cotupling of (insertion) natural transformations

$$\begin{aligned} \kappa &: Var \to T \\ \kappa' &: T \to T \\ \kappa'' &: T^2 \to T \end{aligned}$$

For the first morphism, note that $Var \cong S_1$, so that $\kappa\colon Var \cong S_1 \hookrightarrow T$. It is a simple *Exercise* to check that you understand the definition of κ; do not forget that $S_1 = Var + \varnothing + \varnothing^2$, and so $S_1 n = Var\, n \times \{\, 1 \,\}$. We define κ' by specifying the family of morphisms

$$\kappa'_r\colon S_r \xrightarrow{\quad \iota_S \quad} Var + S_r + S_r^2 = S_{r+1} \hookrightarrow T$$

and appealing to Lemma 4. Note that κ'_r is natural as it is the composition of natural transformations. We must also check that $\kappa'_{r+1} \circ i_r = \kappa'_r$. To do this, we have to check that the following diagram commutes

$$\begin{tikzcd}
S_{r+1} \arrow[r, "\iota_{S\ +1}"] & S_{r+2} \arrow[r, hook] & T \arrow[d, equal] \\
S_r \arrow[u, "i_r"] \arrow[r, "\iota_S"'] & S_{r+1} \arrow[u, "i_{r+1}"] \arrow[r, hook] & T
\end{tikzcd}$$

The right hand square commutes trivially. The left commutes by applying the fact (deduced above) that $i_{r+1} = id_{Var} + i_r + i_r^2$. We can also define κ'' by applying Lemma 4, but the definition requires a little care. Write $S'_r \overset{\text{def}}{=} S_r^2$. Consider the family of morphisms

$$\kappa''_r\colon S'_r = S_r^2 \xrightarrow{\iota_{S^2}} Var + S_r + S_r^2 = S_{r+1} \hookrightarrow T$$

We can check that the κ''_r satisfy the conditions of Lemma 4, and so they define a morphism $\kappa''\colon U \to T$ where $U \overset{\text{def}}{=} \cup_r S'_r$. But note that (why!?)

$$Un = \cup_r S'_r n = \cup_r (S_r n)^2 = (\cup_r S_r n)^2 = (Tn)^2 = T^2 n$$

and one can also check that $U\rho = T^2\rho$. Hence $U = T^2$, and we have our definition of κ''. Thus σ_T is defined in $\mathcal{F}$, as this category has ternary coproducts.

We verify that $\sigma_T : \Sigma_{Var}\, T \to T$ is an initial algebra. Consider

$$\begin{tikzcd}
Var + T + T^2 \arrow[r, "\sigma_T"] \arrow[d, "Var + \overline{\alpha} + \overline{\alpha}^2"'] \arrow[dr, phantom, "(*)"] & T \arrow[d, "\overline{\alpha}"] \\
Var + A + A^2 \arrow[r, "\alpha"] & A
\end{tikzcd}$$

To define $\overline{\alpha}: T \to A$ we specify a family of natural transformations $\overline{\alpha}_r : S_r \to A$ and appeal to Lemma 4. Note that $\overline{\alpha}_0 : \varnothing \to A$ and thus we define $\overline{\alpha}_0$ to be the natural transformation with components the empty functions $\varnothing : \varnothing \to An$ for each n in $\mathbb{F}$. Note that

$$\overline{\alpha}_{r+1} : S_{r+1} = Var + S_r + S_r^2 \to A$$

and hence we can recursively define $\overline{\alpha}_{r+1} \stackrel{\text{def}}{=} [\alpha \circ \iota_{Var}, \alpha \circ \iota_A \circ \overline{\alpha}_r, \alpha \circ \iota_{A^2} \circ \overline{\alpha}_r^2]$. It follows very simply, by induction, that the $\overline{\alpha}_r$ are natural; if $\overline{\alpha}_r$ is natural, then so too is $\overline{\alpha}_{r+1}$, it being the cotupling of compositions of natural transformations. It is an *Exercise* to verify by induction that the conditions of Lemma 4 hold, that is, $\overline{\alpha}_{r+1} \circ i_r = \overline{\alpha}_r$ for all $r \geq 0$.

Using the universal property of finite coproducts, proving that the diagram $(*)$ commutes is equivalent to proving

$$[\overline{\alpha} \circ \kappa, \overline{\alpha} \circ \kappa', \overline{\alpha} \circ \kappa'] = [\alpha \circ \iota_{Var}, \alpha \circ \iota_A \circ \overline{\alpha}, \alpha \circ \iota_{A^2} \circ \overline{\alpha}^2]$$

which in turn is equivalent to proving that the respective components of the cotuples are equal. We prove that $\overline{\alpha} \circ \kappa' = \alpha \circ \iota_A \circ \overline{\alpha}: T \to A$. This amounts to proving that $\overline{\alpha}_n \circ \kappa'_n = \alpha_n \circ \iota_{An} \circ \overline{\alpha}_n : Tn \to An$ in $\mathcal{S}et$ for any n in $\mathbb{F}$. Suppose that ξ is an arbitrary element of Tn, where $\xi \in S_r n$. Then we have

$$\begin{aligned}
\overline{\alpha}_n(\kappa'_n(\xi)) &= \overline{\alpha}_n(\iota_{S\ n}(\xi)) \\
&= (\overline{\alpha}_{r+1})_n(\iota_{S\ n}(\xi)) \\
&= [\alpha_n \circ \iota_{Var\, n}, \alpha_n \circ \iota_{An} \circ (\overline{\alpha}_r)_n, \alpha_n \circ \iota_{(An)^2} \circ (\overline{\alpha}_r)_n^2](\iota_{S\ n}(\xi)) \\
&= \alpha_n(\iota_{An}((\overline{\alpha}_r)_n(\xi))) \\
&= \alpha_n(\iota_{An}(\overline{\alpha}_n(\xi)))
\end{aligned}$$

Each step follows by applying an appropriate function definition.

Step 3. Suppose that $\rho: n \to n'$ is any function. We define

$$Exp_{\mathsf{d\bar{b}}}\, n \stackrel{\text{def}}{=} \{\, e \mid \Gamma^n \vdash^{\mathsf{d\bar{b}}} e \,\}$$

For any expression e there is another expression denoted by $(Exp_{\mathsf{d\bar{b}}}\, \rho)e$ which is, informally, the expression e in which any occurrence of the variable v^i is replaced by $v^{\rho(i)}$. We can define formally the expression $(Exp_{\mathsf{d\bar{b}}}\, \rho)e$ by recursion over e, by setting

- $(\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)(\mathsf{V}\ v^i) \stackrel{\text{def}}{=} \mathsf{V}\ \rho i$
- $(\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)(\mathsf{S}\ e) \stackrel{\text{def}}{=} \mathsf{S}\ (\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)e$
- $(\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)(\mathsf{A}\ e\ e') \stackrel{\text{def}}{=} \mathsf{A}\ (\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)e\ (\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)e'$

Further, one can show that if $e \in \mathit{Exp}_{\mathsf{d\bar{b}}}\ n$, then $(\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho)e \in \mathit{Exp}_{\mathsf{d\bar{b}}}\ n'$. Thus we have a function $\mathit{Exp}_{\mathsf{d\bar{b}}}\ \rho : \mathit{Exp}_{\mathsf{d\bar{b}}}\ n \to \mathit{Exp}_{\mathsf{d\bar{b}}}\ n'$ for any $\rho : n \to n'$. It is an *Exercise* to verify that these definitions yield a functor $\mathit{Exp}_{\mathsf{d\bar{b}}} : \mathbb{F} \to \mathcal{S}et$. Further, note that there are natural transformations (*Exercise*) $\mathsf{S}: \mathit{Exp}_{\mathsf{d\bar{b}}} \to \mathit{Exp}_{\mathsf{d\bar{b}}}$ and $\mathsf{A}: {\mathit{Exp}_{\mathsf{d\bar{b}}}}^2 \to \mathit{Exp}_{\mathsf{d\bar{b}}}$ whose obvious definitions are omitted.

Step 4. We have constructed an initial algebra $\sigma_T : \Sigma_{\mathit{Var}}\ T \to T$ in the category $\mathcal{F}$. We now show that the presheaf algebra T is isomorphic to the presheaf $\mathit{Exp}_{\mathsf{d\bar{b}}}$. To do this we need natural transformations $\phi : T \to \mathit{Exp}_{\mathsf{d\bar{b}}}$ and $\psi : \mathit{Exp}_{\mathsf{d\bar{b}}} \to T$, such that for any n in $\mathbb{F}$, the functions ϕ_n and ψ_n give rise to a bijection between Tn and $\mathit{Exp}_{\mathsf{d\bar{b}}}\ n$.

To specify $\phi : T \to \mathit{Exp}_{\mathsf{d\bar{b}}}$ we define a family of natural transformations $\phi_r : S_r \to \mathit{Exp}_{\mathsf{d\bar{b}}}$, and appeal to Lemma 4.

- $\phi_0 : S_0 = \varnothing \to \mathit{Exp}_{\mathsf{d\bar{b}}}$ has the empty function as components $(\phi_0)_n : \varnothing \to \mathit{Exp}_{\mathsf{d\bar{b}}}\ n$
- Recursively we define

$$\phi_{r+1} \stackrel{\text{def}}{=} [\mathsf{V}, \mathsf{S} \circ \phi_r, \mathsf{A} \circ \phi_r^2] : S_{r+1} = \mathit{Var} + S_r + S_r^2 \to \mathit{Exp}_{\mathsf{d\bar{b}}}$$

Note that ϕ_0 is obviously natural, and that ϕ_{r+1} is natural if ϕ_r is, because $\mathcal{F}$ has ternary coproducts. It is an *Exercise* to verify the conditions of the lemma.

To specify $\psi : \mathit{Exp}_{\mathsf{d\bar{b}}} \to T$, for any n in $\mathbb{F}$ we define functions $\psi_n : \mathit{Exp}_{\mathsf{d\bar{b}}}\ n \to Tn$ as follows. First note that $S_r n \subset Tn$ for any r by definition of T. Then we define

- $\psi_n(\mathsf{V}\ v^i) \stackrel{\text{def}}{=} (v^i, 1) \in S_1 n$
- $\psi_n(\mathsf{S}\ e) \stackrel{\text{def}}{=} \iota_{S\ n}(\psi_n(e))$ where $r \geq 1$ is the height of the deduction of $\mathsf{S}\ e$
- $\psi_n(\mathsf{A}\ e\ e') \stackrel{\text{def}}{=} \iota_{(S\ n)^2}((\psi_n(e), \psi_n(e')))$ where $r \geq 1$ is the height of the deduction of $\mathsf{A}\ e\ e'$.

We next check that for any n in $\mathbb{F}$,

$$\begin{tikzcd} Tn \arrow[r, shift left, "\phi_n"] \arrow[r, phantom, "\cong"] & \mathit{Exp}_{\mathsf{d\bar{b}}}\ n \arrow[l, shift left, "\psi_n"] \end{tikzcd}$$

We need the lemma

Lemma 5. *For any $r \geq 0$, and $\xi \in S_{r+1}n$, the expression $(\phi_{r+1})_n(\xi)$ has a deduction of height r.*

Proof. By induction on r.

Suppose that $\xi \in S_r n \subset Tn$ for some r. Then by definition, $\psi_n(\phi_n(\xi)) = \psi_n((\phi_r)_n(\xi))$. We show by induction that for all $r \geq 0$, if ξ is any element of $S_r n$ and n any object of $\mathbb{F}$, then $\psi_n((\phi_r)_n(\xi)) = \xi$. For $r = 0$ the assertion is vacuously true, as $S_0 n$ is always empty. We assume the result holds for any $r \geq 0$. Let $\xi \in S_{r+1}n = \mathit{Var}\, n + S_r n + S_r n^2$. Then we have

$$\psi_n((\phi_{r+1})_n(\xi)) = \psi_n([\mathsf{V}_n, \mathsf{S}_n \circ (\phi_r)_n, \mathsf{A}_n \circ (\phi_r)^2_n](\xi))$$

We can complete the proof by case analysis. The situation $r = 0$ requires a little care, but we leave it as an *Exercise*. We just consider the case when $\xi = \iota_{S\ n}(\xi')$ for some $\xi' \in S_r n$ (which implies that $r \geq 1$). We have

$$\begin{aligned}
\psi_n((\phi_{r+1})_n(\xi)) &= \psi_n((\mathsf{S}_n \circ (\phi_r)_n)(\xi')) && (1)\\
&= \psi_n(\mathsf{S}\ (\phi_r)_n(\xi')) && (2)\\
&= \iota_{S\ n}(\psi_n((\phi_r)_n(\xi'))) && (3)\\
&= \iota_{S\ n}(\xi') && (4)\\
&= \xi && (5)
\end{aligned}$$

where equation 3 follows from Lemma 5, and equation 4 by induction. It is an *Exercise* to show that ϕ_n is a left inverse for ψ_n. By appeal to Lemma 1 we are done, and step 4 is complete.

However, by way of illustration, we can check directly by hand that ψ is natural, that is for any $\rho\colon n \to n'$ the diagram

$$\begin{tikzcd}
\mathit{Exp}_{\mathsf{d\bar{b}}}\, n \arrow[r, "\psi_n"] \arrow[d, "\mathit{Exp}_{\mathsf{d\bar{b}}}\, \rho"'] & Tn \arrow[d, "T\rho"] \\
\mathit{Exp}_{\mathsf{d\bar{b}}}\, n' \arrow[r, "\psi'_n"'] & Tn'
\end{tikzcd}$$

commutes. We must prove that for all $\Gamma^n \vdash^{\mathsf{d\bar{b}}} e$, we have

$$T\rho(\psi_n(e)) = \psi_{n'}((\mathit{Exp}_{\mathsf{d\bar{b}}}\, \rho)(e))$$

We consider only the inductive case of $\mathsf{S}\ e$:

$$\begin{aligned}
T\rho(\psi_n(\mathsf{S}\ e)) &= T\rho(\iota_{S\ n}(\psi_n(e)))\\
&= S_{r+1}\rho(\iota_{S\ n}(\psi_n(e)))\\
&= (\mathit{Var}\,\rho + S_r\rho + (S_r\rho)^2)((\iota_{S\ n}(\psi_n(e))))\\
&= \iota_{S\ n}((S_r\rho)(\psi_n(e)))\\
&= \iota_{S\ n}((T\rho)(\psi_n(e)))\\
&= \iota_{S\ n'}((\psi'_n(\mathit{Exp}_{\mathsf{d\bar{b}}}\, \rho(e))))\\
&= \psi_{n'}(\mathsf{S}\ (\mathit{Exp}_{\mathsf{d\bar{b}}}\, \rho)(e))\\
&= \psi_{n'}(\mathit{Exp}_{\mathsf{d\bar{b}}}\, \rho(\mathsf{S}\ e))
\end{aligned}$$

It is an *Exercise* to check this calculation; the sixth equality follows by induction.

4.2 A Model of Syntax with Distinguished Variables and with Binding

Step 1. We define a functor which "corresponds" to the signature of Section 2.2. This is $\Sigma_{Var} : \mathcal{F} \to \mathcal{F}$ where $\Sigma_{Var}\, \xi \overset{\text{def}}{=} Var + \delta\, \xi + \xi^2$ and ξ is either an object or a morphism. The functor $\delta : \mathcal{F} \to \mathcal{F}$ was defined on page 156.

Step 2. We can show that the functor Σ_{Var} has an initial algebra, which we denote by $\sigma_T : \Sigma_{Var}\, T \to T$, by adapting the methods given in Section 4.1. In fact there is very little change in the details, so we just sketch the general approach and leave the technicalities as an *Exercise.*

We define a family of presheaves $(S_r \mid r \geq 0)$, such that we may apply Lemma 3. We set $S_0 \overset{\text{def}}{=} \varnothing$ which is the empty presheaf, and set $S_{r+1} \overset{\text{def}}{=} \Sigma_{Var}\, S_r$. We must check that the conditions of Lemma 3 hold, that is, $i_r : S_r \hookrightarrow S_{r+1}$ for all $r \geq 0$. This can be done by induction over r, and is left as an *exercise.* We can then define $T \overset{\text{def}}{=} \bigcup_r S_r$ in $\mathcal{F}$.

Next we consider the structure map σ_T. This natural transformation in $\mathcal{F}$ has source and target $Var + \delta\ T + T^2 \to T$ and so it is given by $\sigma_T \overset{\text{def}}{=} [\kappa, \kappa', \kappa'']$, the cotupling of (insertion) natural transformations

$$\begin{aligned} \kappa &: Var \to T \\ \kappa' &: \delta\ T \to T \\ \kappa'' &: T^2 \to T \end{aligned}$$

The definitions of κ and κ'' are the same as in Section 4.1. Note that $(\delta\ T)n \overset{\text{def}}{=} T(n+1) = \bigcup_r S_r(n+1) = \bigcup_r(\delta\ S_r)n = (\bigcup_r \delta\ S_r)n$. In fact we can easily check that $\delta\ T = \cup_r \delta\ S_r$, as similar equalities hold if we replace n by any ρ. Hence we can apply an instance of Lemma 4 to give a definition of κ' by specifying the family of morphisms κ'_r with the following definition

$$\kappa'_r : \delta\ S_r \xrightarrow{\iota_S} Var + \delta\ S_r + S_r^2 = S_{r+1} \hookrightarrow T$$

Note that we must check to see that $\delta\ S_r \hookrightarrow \delta\ S_{r+1}$ for all r. Use induction to verify this as an *Exercise*; note that for any $F' \hookrightarrow F$ in $\mathcal{F}$, we have $\delta\ F' \hookrightarrow \delta\ F$.

We must verify that $\sigma_T : \Sigma_{Var}\ T \to T$ is an initial algebra. Consider

$$\begin{tikzcd} Var + \delta\ T + T^2 \arrow[r, "\sigma_T"] \arrow[d, "Var + \delta\,\overline{\alpha} + \overline{\alpha}^2"'] \arrow[dr, phantom, "(*)"] & T \arrow[d, "\overline{\alpha}"] \\ Var + \delta\ A + A^2 \arrow[r, "\alpha"'] & A \end{tikzcd}$$

To define $\overline{\alpha} : T \to A$ we specify a family of natural transformations $\overline{\alpha}_r : S_r \to A$ and appeal to Lemma 4. We define $\overline{\alpha}_0$ to be the natural transformation with components the empty functions $\varnothing : \varnothing \to An$ for each n in $\mathbb{F}$, and

$$\overline{\alpha}_{r+1} \overset{\text{def}}{=} [\alpha \circ \iota_{Var}, \alpha \circ \iota_A \circ \delta\, \overline{\alpha}_r, \alpha \circ \iota_{A^2} \circ \overline{\alpha}_r^2] \colon S_{r+1} = Var + S_r + S_r^2 \to A$$

The verification that $(*)$ commutes is technically identical to the analogous situation in Section 4.1, and the details are left as an *Exercise.*

Step 3. Suppose that $\rho\colon n \to n'$. We write $Exp_{\mathsf{db}}\, n$ for the set $\{\, e \mid \Gamma^n \vdash^{\mathsf{db}} e \,\}$ of expressions deduced using the rules in Figure 2. Let $\rho\{n'/n\}\colon n+1 \to n'+1$ be the function

$$\rho\{n'/n\}(j) \overset{\text{def}}{=} \begin{cases} j & \text{if} \quad 0 \le j \le n-1 \\ n' & \text{if} \quad j = n \end{cases}$$

Consider the following (syntactic) definitions

– $(Exp_{\mathsf{db}}\, \rho)(\mathsf{V}\ v^i) \overset{\text{def}}{=} \mathsf{V}\ v^{\rho i}$
– $(Exp_{\mathsf{db}}\, \rho)(\mathsf{L}\ v^n\ e) \overset{\text{def}}{=} \mathsf{L}\ v^{n'}\ (Exp_{\mathsf{db}}\, \rho\{n'/n\})(e)$ and
– $(Exp_{\mathsf{db}}\, \rho)(\mathsf{E}\ e\ e') \overset{\text{def}}{=} \mathsf{E}\ ((Exp_{\mathsf{db}}\, \rho)e)\ ((Exp_{\mathsf{db}}\, \rho)e')$

One can prove by rule induction that if $\Gamma^n \vdash^{\mathsf{db}} e$ and $\rho\colon n \to n'$, then $\Gamma^{n'} \vdash^{\mathsf{db}} (Exp_{\mathsf{db}}\ \rho)e$. In fact we have a functor Exp_{db} in $\mathcal{F}$ and the details are an *Exercise.* Note also that there are natural transformations $\mathsf{L}\colon \delta\ Exp_{\mathsf{db}} \to Exp$ and $\mathsf{E}\colon Exp^2 \to Exp$, *provided certain assumptions are made about the category* $\mathbb{F}$!! The latter's definition is the obvious one. For the former, the components are functions $\mathsf{L}_n\colon Exp_{\mathsf{db}}\ (n+1) \to Exp_{\mathsf{db}}\ n$ defined by $e \mapsto \mathsf{L}\ v^n\ e$. Naturality is the requirement that for any $\rho\colon n \to n'$ in $\mathbb{F}$, the diagram below commutes

$$\begin{tikzcd}
(\delta\ Exp_{\mathsf{db}})n = Exp_{\mathsf{db}}\ (n+1) \arrow[r, "\mathsf{L}_n"] \arrow[d, "(\delta\ Exp_{\mathsf{db}})\rho = Exp_{\mathsf{db}}\ (\rho + id_1)"'] & Exp_{\mathsf{db}}\ n \arrow[d, "Exp_{\mathsf{db}}\ \rho"] \\
(\delta\ Exp_{\mathsf{db}})n' = Exp_{\mathsf{db}}\ (n'+1) \arrow[r, "\mathsf{L}_{n'}"'] & Exp_{\mathsf{db}}\ n'
\end{tikzcd}$$

Note that at the element e, this requires that

$$\mathsf{L}\ v^{n'}\ (Exp_{\mathsf{db}}\ \rho\{n'/n\})e = \mathsf{L}\ v^{n'}\ ((Exp_{\mathsf{db}}\ (\rho + id_1))e)$$

and by considering when e is a variable, we conclude that this equality holds if and only if

$$\rho\{n'/n\} = \rho + id_1$$

which is true only if in $\mathbb{F}$ the coproduct insertion $\iota_1\colon 1 \to m+1$ maps $*$ to m, and $\iota_m\colon m \to m+1$ maps i to ρi for any $i \in m$.

Step 4. We now show that the presheaf algebra T is isomorphic to the presheaf $\mathit{Exp}_{\mathsf{db}}$. We have to show that there are natural transformations $\phi\colon T \to \mathit{Exp}_{\mathsf{db}}$ and $\psi\colon \mathit{Exp}_{\mathsf{db}} \to T$, such that for any n in $\mathbb{F}$, the functions ϕ_n and ψ_n give rise to a bijection between Tn and $\mathit{Exp}_{\mathsf{db}}\, n$.

To specify $\phi\colon T \to \mathit{Exp}_{\mathsf{db}}$ we define a family of natural transformations $\phi_r\colon S_r \to \mathit{Exp}_{\mathsf{db}}$, and appeal to Lemma 4.

– $\phi_0\colon S_0 = \varnothing \to \mathit{Exp}_{\mathsf{db}}$ has as components the empty function, and
– recursively we define

$$\phi_{r+1} \overset{\text{def}}{=} [\mathsf{V}, \mathsf{L} \circ \delta\, \phi_r, \mathsf{E} \circ \phi_r^2]\colon S_{r+1} = \mathit{Var} + \delta\, S_r + S_r^2 \to \mathit{Exp}_{\mathsf{db}}$$

To specify $\psi\colon \mathit{Exp}_{\mathsf{db}} \to T$, for any n in $\mathbb{F}$ we define functions $\psi_n\colon \mathit{Exp}_{\mathsf{db}}\, n \to Tn$ as follows. First note that $S_r n \subset Tn$ for any r by definition of T. Then we define

– $\psi_n(\mathsf{V}\ v^i) \overset{\text{def}}{=} (v^i, 1) \in S_1 n$.
– $\psi_n(\mathsf{L}\ v^n\ e) \overset{\text{def}}{=} \iota_{S\ (n+1)}(\psi_{n+1}(e))$ where $r \geq 0$ is the height of the deduction of $\mathsf{L}\ v^n\ e$.
– $\psi_n(\mathsf{E}\ e\ e') \overset{\text{def}}{=} \iota_{(S\ n)^2}((\psi_n(e), \psi_n(e')))$ where $r \geq 0$ is the height of the deduction of $\mathsf{E}\ e\ e'$.

We check also that for any n in $\mathbb{F}$,

$$\begin{tikzcd} Tn \arrow[r, shift left, "\phi_n"] \arrow[r, phantom, "\cong"] & \mathit{Exp}_{\mathsf{db}}\, n \arrow[l, shift left, "\psi_n"] \end{tikzcd}$$

Suppose that $\xi \in S_r n \subset Tn$. Then by definition, $\psi_n(\phi_n(\xi)) = \psi_n((\phi_r)_n(\xi))$. We show by induction that for all $r \geq 0$, if ξ is any element in level r and n any object of $\mathbb{F}$, then $\psi_n((\phi_r)_n(\xi)) = \xi$. For $r = 0$ the assertion is vacuously true, as $S_0 n$ is always empty. We assume the result holds for any $r \geq 0$. Let $\xi \in S_{r+1} n = \mathit{Var}\, n + S_r(n+1) + S_r n^2$. Then we have

$$\psi_n((\phi_{r+1})_n(\xi)) = \psi_n([\mathsf{V}_n, \mathsf{L}_n \circ (\phi_r)_{n+1}, \mathsf{E}_n \circ (\phi_r)_n^2](\xi))$$

We can complete the proof by analyzing the cases which arise depending on which component ξ lives in. We just consider the case when $\xi = \iota_{S\ (n+1)}(\xi')$ for some $\xi' \in S_r(n+1)$. We have

$$\begin{aligned} \psi_n((\phi_{r+1})_n(\xi)) &= \psi_n((\mathsf{L}_n \circ (\phi_r)_{n+1})(\xi')) && (6)\\ &= \psi_n(\mathsf{L}\ v^n\ (\phi_r)_{n+1}(\xi')) && (7)\\ &= \iota_{S\ (n+1)n}(\psi_{n+1}((\phi_r)_{n+1}(\xi'))) && (8)\\ &= \iota_{S\ (n+1)n}(\xi') && (9)\\ &= \xi && (10)\\ & && (11) \end{aligned}$$

where equation 8 follows from Lemma 5, and equation 9 by induction. It is an *Exercise* to show that ϕ_n is a left inverse for ψ_n, and we are done appealing to Lemma 1.

By way of illustration, we give a sample of the direct calculation that each ψ_n is natural. We prove that for all $\Gamma^n \vdash^{\mathsf{db}} e$, we have

$$T\rho(\psi_n(e)) = \psi_{n'}((\mathit{Exp}_{\mathsf{db}}\,\rho)(e))$$

We consider the case of $\mathsf{L}\ v^n\ e$; checking the details is an *Exercise.*

$$\begin{aligned}
T\rho(\psi_n(\mathsf{L}\ v^n\ e)) &= T\rho(\iota_{S\ (n+1)}(\psi_{n+1}(e)))\\
&= S_{r+1}\rho(\iota_{S\ (n+1)}(\psi_{n+1}(e)))\\
&= ((\mathit{Var}\,\rho + (\delta\ S_r)(\rho) + (S_r\rho)^2)\rho)(\iota_{S\ (n+1)}(\psi_{n+1}(e)))\\
&= ((\mathit{Var}\,\rho + S_r(\rho + \mathit{id}_1) + (S_r\rho)^2)\rho)(\iota_{S\ (n+1)}(\psi_{n+1}(e)))\\
&= \iota_{S\ (n'+1)}((S_r(\rho + \mathit{id}_1))(\psi_{n+1}(e)))\\
&= \iota_{S\ (n'+1)}((T(\rho + \mathit{id}_1))(\psi_{n+1}(e)))\\
&= \iota_{S\ (n'+1)}((\psi_{n'+1}(\mathit{Exp}_{\mathsf{db}}\ (\rho + \mathit{id}_1)(e)))\\
&= \iota_{S\ (n'+1)}(\psi_{n'+1}((\mathit{Exp}_{\mathsf{db}}\,\rho\{n'/n\})e))))\\
&= \psi_{n'}(\mathsf{L}\ v^{n'}\ (\mathit{Exp}_{\mathsf{db}}\,\rho\{n'/n\})(e))\\
&= \psi_{n'}((\mathit{Exp}_{\mathsf{db}}\,\rho)(\mathsf{L}\ v^n\ e))
\end{aligned}$$

4.3 A Model of Syntax with Arbitrary Variables and Binding

We define a functor which "corresponds" to the signature of Section 2.3. The functor $\Sigma_{\mathit{Var}} : \mathcal{F} \to \mathcal{F}$ is defined by setting $\Sigma_{\mathit{Var}}\,\xi \overset{\mathrm{def}}{=} \mathit{Var} + \delta\,\xi + \xi^2$. As you see, it is identical to the functor given at the start of Section 4.2, and thus has an initial algebra $\sigma_T : \Sigma_{\mathit{Var}}\ T \to T$.

We show in this section that we can define a functor $\mathit{Exp}_{\mathsf{ab}}$ in $\mathcal{F}$ which captures the essence of the inductive system of expressions given in Section 2.3 and is such that $\mathit{Exp}_{\mathsf{ab}} \cong T$. We could prove this by proceeding (directly) as we did in Section 4, undertaking the steps 2 to 4 of page 155. However, it is in fact easier, and more instructive, to first define $\mathit{Exp}_{\mathsf{ab}}$, step 3, and then prove that $\mathit{Exp}_{\mathsf{ab}} \cong \mathit{Exp}_{\mathsf{db}}$. Given previous results, this gives us steps 2 and 4.

We will need two lemmas which yield *admissible* rules (see Appendix). The rules cannot be derived.

Lemma 6. *Suppose that $\Delta \vdash^{\mathsf{ab}} e$, and that Δ' is also an environment. Then $\Delta' \vdash^{\mathsf{ab}} e\{\Delta'/\Delta\}$.*

Proof. We prove by rule induction

$$(\forall \Delta \vdash^{\mathsf{ab}} e) \quad (\forall \Delta')\ (\Delta' \vdash^{\mathsf{ab}} e\{\Delta'/\Delta\})$$

We prove property closure for the rule introducing abstractions $\mathsf{L}\ x\ e$. Suppose that $\Delta \vdash^{\mathsf{ab}} \mathsf{L}\ x\ e$. Then $\Delta, x \vdash^{\mathsf{ab}} e$. Pick any Δ'. We try to prove that

$$\Delta' \vdash^{\mathsf{ab}} (\mathsf{L}\ x\ e)\{\Delta'/\Delta\}$$

We consider only the case when $x \in \Delta'$ at position p, and $y \overset{\text{def}}{=} \mathsf{el}_p(\Delta) \in fv(e)$. We must then prove

$$\Delta' \vdash^{\mathsf{ab}} \mathsf{L}\ x'\ e\{\Delta', x'/\Delta, x\} \qquad *$$

which is well-defined as $x \notin \Delta$ and $x' \notin \Delta'$. By induction, we have $\Delta', x' \vdash^{\mathsf{ab}} e\{\Delta', x'/\Delta, x\}$. Hence $*$ follows.

Lemma 7. *If $\Delta \vdash^{\mathsf{ab}} e$, and Δ is a sublist of an environment Δ_1, then $\Delta_1 \vdash^{\mathsf{ab}} e$.*

Proof. Rule Induction. *Exercise.*

Back to the task at hand. First we must define Exp_{ab} . For n in $\mathbb{F}$ we set

$$Exp_{\mathsf{ab}}\ n \overset{\text{def}}{=} \{\ [e]_\alpha \ \mid\ \Gamma^n \vdash^{\mathsf{ab}} e\ \}$$

Now let $\rho\colon n \to n'$. We define

$$(Exp_{\mathsf{ab}}\ \rho)([e]_\alpha) \overset{\text{def}}{=} [e\{v^{\rho 0}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\}]_\alpha$$

To check if this is a good definition, we need to show that if $\Gamma^n \vdash^{\mathsf{ab}} e$ then

$$\Gamma^{n'} \vdash^{\mathsf{ab}} e\{v^{\rho 0}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\}$$

This follows from Lemma 6 and Lemma 7. We must also check that if there is any e' with $e \sim_\alpha e'$, then

$$e\{v^{\rho 0}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\} \sim_\alpha e'\{v^{\rho 0}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\}$$

This is proved by rule induction for $\sim_\alpha$, a tedious *Exercise.*

We now show that $\phi\colon Exp_{\mathsf{ab}} \cong Exp_{\mathsf{db}} : \psi$. The components of ψ are functions $\psi_n\colon Exp_{\mathsf{db}}\ n \to Exp_{\mathsf{ab}}\ n$ given by $\psi_n(e) \overset{\text{def}}{=} [e]_\alpha$. We consider the naturality of ψ at a morphism $\rho\colon n \to n'$, computed at an element ξ of $Exp_{\mathsf{db}}\ n$. We show naturality for the case $\xi = \mathsf{L}\ v^n\ e$.

$$\begin{aligned}(Exp_{\mathsf{ab}}\ \rho) \circ \psi_n(\xi) &= (Exp_{\mathsf{ab}}\ \rho)[\mathsf{L}\ v^n\ e]_\alpha \\ &= [(\mathsf{L}\ v^n\ e)\{v^{\rho 0}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\}]_\alpha \\ &\overset{\text{def}}{=} \square\end{aligned}$$

Let us consider the case when renaming takes place. Suppose that there is a j for which $\rho(j) = n$ and $v^j \in fv(e)$. Then[6]

$$\begin{aligned}&(\mathsf{L}\ v^n\ e)\{v^{\rho(0)}, \ldots, v^{\rho(n-1)}/v^0, \ldots, v^{n-1}\} = \\ &\qquad \mathsf{L}\ v^w\ e\{v^{\rho(0)}, \ldots v^{\rho(n-1)}, v^w/v^0, \ldots, v^{n-1}, v^n\}\end{aligned}$$

where w is 1 plus the maximum of the indices occurring freely in e and the indices $\rho(0), \ldots, \rho(n-1)$. Thus $\rho(i) < w$ for all $0 \le i \le n-1$. But the free variables in e

[6] There is no deletion.

must lie in $v^0, \ldots, v^n$ (why?) and moreover $n = \rho(j)$ occurs in $\rho(0), \ldots, \rho(n-1)$. Finally note that $\rho(i) < n'$, and so we must have $w \leq n'$. If $w < n'$, then $v^{n'}$ is not free in $e\{v^{\rho(0)}, \ldots v^{\rho(n-1)}, v^w/v^0, \ldots, v^{n-1}, v^n\}$. Otherwise (of course) $w = n'$. Either way (why!?),

$$\begin{aligned}&\mathsf{L}\ v^w\ e\{v^{\rho(0)}, \ldots v^{\rho(n-1)}, v^w/v^0, \ldots, v^{n-1}, v^n\}\\ &\quad\sim_\alpha \mathsf{L}\ v^{n'}\ e\{v^{\rho(0)}, \ldots v^{\rho(n-1)}, v^{n'}/v^0, \ldots, v^{n-1}, v^n\}\end{aligned}$$

and so

$$\begin{aligned}\square &= [\mathsf{L}\ v^{n'}\ e\{v^{\rho 0}, \ldots, v^{\rho(n-1)}, v^{n'}/v^0, \ldots, v^{n-1}, v^n\}]_\alpha\\ &= [\mathsf{L}\ v^{n'}\ (\mathit{Exp}_{\mathsf{db}}\ \rho\{n'/n\})e]_\alpha\\ &= \psi_{n'} \circ (\mathit{Exp}_{\mathsf{db}}\ \rho)(\xi)\end{aligned}$$

Next we define the functions $\phi_n\colon \mathit{Exp}_{\mathsf{ab}}\ n \to \mathit{Exp}_{\mathsf{db}}\ n$ by setting $\phi_n([e]_\alpha) \overset{\text{def}}{=} R^n(e)$ where

– $R^m(\mathsf{V}\ x) \overset{\text{def}}{=} \mathsf{V}\ x$
– $R^m(\mathsf{L}\ x\ e) \overset{\text{def}}{=} \mathsf{L}\ v^m\ R^{m+1}(e\{v^m/x\})$
– $R^m(\mathsf{E}\ e\ e') \overset{\text{def}}{=} \mathsf{E}\ R^m(e)\ R^m(e')$

To be well-defined, we require $R^m(e) = R^m(e')$ for all $e \sim_\alpha e'$. We can prove this by induction over $\sim_\alpha$. The only tricky case concerns the axiom for re-naming, $\mathsf{L}\ x\ e \sim_\alpha \mathsf{L}\ x'\ e\{x'/x\}$ where $x' \notin \mathit{fv}(e)$. We have

$$\begin{aligned}R^m(\mathsf{L}\ x'\ e\{x'/x\}) &= \mathsf{L}\ v^m\ R^{m+1}(e\{x'/x\}\{v^m/x'\})\\ &= \mathsf{L}\ v^m\ R^{m+1}(e\{v^m/x\})\\ &= R^m(\mathsf{L}\ x\ e)\end{aligned}$$

with the second equality holding as $x' \notin \mathit{fv}(e)$. Let $e \in \mathit{Exp}_{\mathsf{db}}\ n$. We will have $\phi_n(\psi_n(e)) = e$ provided that $R^n(e) = e$. We can prove this by rule induction, showing

$$(\forall \Gamma^n \vdash^{\mathsf{db}} e)(R^n(e) = e)$$

The details are easy and left as an *Exercise*. Let $e \in \mathit{Exp}_{\mathsf{ab}}\ n$. It remains to prove that $e = \psi_n(\phi_n(e))$. This will hold provided that $R^n(e) \sim_\alpha e$. We prove

$$(\forall \Delta \vdash^{\mathsf{ab}} e) \quad (\forall \Gamma^n)\ (\Delta = \Gamma^n \implies R^n(e) \sim_\alpha e)$$

We show property closure for the rule introducing abstractions. Suppose that $\Gamma^n \vdash^{\mathsf{ab}} \mathsf{L}\ x\ e$. We must show that $R^n(\mathsf{L}\ x\ e) \sim_\alpha \mathsf{L}\ x\ e$. Now $\Gamma^n, x \vdash^{\mathsf{ab}} e$ and so by a *careful* use of Lemma 6 we get $\Gamma^{n+1} \vdash^{\mathsf{ab}} e\{v^n/x\}$. Hence by induction $R^{n+1}(e\{v^n/x\}) \sim_\alpha e\{v^n/x\}$, and so

$$R^n(\mathsf{L}\ x\ e) \overset{\text{def}}{=} \mathsf{L}\ v^n\ R^{n+1}(e\{v^n/x\}) \sim_\alpha \mathsf{L}\ v^n\ e\{v^n/x\}$$

If $x = v^n$ we are done. If not, noting that $x \notin \Gamma^n$ by assumption, $v^n \notin \mathit{fv}(e)$. Hence $\mathsf{L}\ v^n\ e\{v^n/x\} \sim_\alpha \mathsf{L}\ x\ e$.

4.4 A Model of Syntax without Variables but with Binding

Again, we show how some syntax, this time the de Bruijn expressions of Section 2.4, can be rendered as a functor. We give only bare details, and leave most of the working to the reader. We do assume that readers are already familiar with the de Bruijn notation.

For any n in $\mathbb{F}$ we define $\mathit{Exp}_{\mathsf{ib}}\, n \overset{\text{def}}{=} \{\, e \mid n \vdash^{\mathsf{ib}} e \,\}$. We define for $\rho\colon n \to n'$ the function $\mathit{Exp}_{\mathsf{ib}}\, \rho$ by recursion. Consider the following (syntactic) definitions

- $(\mathit{Exp}_{\mathsf{ib}}\, \rho)(\mathsf{V}\ i) \overset{\text{def}}{=} \mathsf{V}\ \rho i$
- $(\mathit{Exp}_{\mathsf{ib}}\, \rho)(\lambda\ e) \overset{\text{def}}{=} \lambda\ (\mathit{Exp}_{\mathsf{ib}}\, \rho^*)(e)$ and
- $(\mathit{Exp}_{\mathsf{ib}}\, \rho)(\$\ e\ e') \overset{\text{def}}{=} \$\ ((\mathit{Exp}_{\mathsf{ib}}\, \rho)e)\ ((\mathit{Exp}_{\mathsf{ib}}\, \rho)e')$

where $\rho^*\colon n{+}1 \to n'{+}1$ and $\rho^*(0) \overset{\text{def}}{=} 0$ and $\rho^*(i{+}1) \overset{\text{def}}{=} \rho(i){+}1$ for $0 \le i \le n{-}1$. One can prove by induction that if $n \vdash^{\mathsf{ib}} e$ then $n' \vdash^{\mathsf{ib}} (\mathit{Exp}_{\mathsf{ib}}\, \rho)e$, and thus $\mathit{Exp}_{\mathsf{ib}}\, \rho$ is well-defined.

One can prove that $\mathit{Exp}_{\mathsf{ib}} \cong T$ by adapting the methods of Section 4.2, establishing a natural isomorphism $\phi\colon T \cong \mathit{Exp}_{\mathsf{ib}} : \psi$. Such an isomorphism exists only for a specific choice of coproducts in $\mathbb{F}$.

To specify $\phi\colon T \to \mathit{Exp}_{\mathsf{ib}}$ we define a family of natural transformations $\phi_r\colon S_r \to \mathit{Exp}_{\mathsf{ib}}$, and appeal to Lemma 4, as follows.

- $\phi_0\colon S_0 = \varnothing \to \mathit{Exp}_{\mathsf{ib}}$ has as components the empty function, and
- recursively we define

$$\phi_{r+1} \overset{\text{def}}{=} [\mathsf{V}, \lambda \circ \delta\ \phi_r, \$ \circ \phi_r^2]\colon S_{r+1} = \mathit{Var} + \delta\ S_r + S_r^2 \to \mathit{Exp}_{\mathsf{ib}}$$

where there are natural transformations $\lambda\colon \mathit{Exp}_{\mathsf{ib}} \to \mathit{Exp}_{\mathsf{ib}}$ and $\$\colon (\mathit{Exp}_{\mathsf{ib}})^2 \to \mathit{Exp}_{\mathsf{ib}}$. Of course $\lambda_n(e) \overset{\text{def}}{=} \lambda\ e$ for any e in $\mathit{Exp}_{\mathsf{ib}}\, n$. This is natural only if the coproduct insertion $\iota_1\colon 1 \to m+1$ maps $*$ to 0, and $\iota_m\colon m \to m+1$ maps $i+1$ to $\rho(i)+1$ for any $0 \le i \le m-1$.

We leave the definition of ψ, and all other calculations, as a long(ish) *Exercise.*

4.5 Where to Now?

There are a number of books which cover basic category theory. For a short and gentle introduction, see [27]. For a longer first text see [4]. Both of these books are intended for computer scientists. The original and recommended general reference for category theory is [19], which was written for mathematicians. A very concise and fast paced introduction can be found in [10] which also covers the theory of allegories (which, roughly, are to relations, what categories are to functions). Again for the more advanced reader, try [31] which is an essential read for anyone interested in categorical logic, and which has a lot of useful background information. The Handbook of Logic in Computer Science has a wealth of material which is related to categorical logic; there is a chapter [29] on category theory. Equalities such as those that arise from universal properties can often be established using so-called calculational methods. For a general introduction, and more references, see [2]. Finally we mention [32] which, apart from being a very interesting introduction to category theory due to the many and varied computing examples, has a short chapter devoted to distributive categories.

The material in these notes has its origins in the paper [9]. You will find that these notes provide much of the detail which is omitted from the first few sections of this paper. In addition, you will find an interesting abstract account of substitution, and a detailed discussion of initial algebra semantics.

The material in these notes, and in [9], is perhaps rather closely allied to the methodology of de Bruijn expressions. Indeed, in these notes, when we considered λ-calculus, we either introduced α-equivalence, or we had a system of expressions which forced a binding variable to be v^n whenever the free variables of the subexpression of an abstraction were in Γ^n. We would like to be able to define a datatype whose elements are the expressions of the λ-calculus, already identified up to α-equivalence. This is achieved by Pitts and Gabbay [11], who undertake a fundamental study of α-equivalence, and formulate a new set theory within which this is possible. For further work see [28] and the references therein.

There is a wealth of literature on so called Higher Order Abstract Syntax. This is another methodology for encoding syntax with binders. For an introduction, see [26, 25], although the ideas go back to Church. The paper [14] provides links between Higher Order Abstract Syntax, and the presheaf models described in these notes. For material on implementation, see [7, 8]. A more recent approach, which combines de Bruijn notation and ordinary λ-calculus in a *hybrid* syntax, is described in [1].

If you are interested in direct implementations of α-equivalence, see [12, 13]. See [6] for the origins of de Bruijn notation.

The equation $E \cong V + E + (E \times E)$ is a very simple example of a *domain* equation. Such equations arise frequently in the study of the semantics of programming languages. They do not always have solutions in $\mathcal{S}et$. However, many can be solved in other categories. See for example [30]. Readers should note that the lemmas given in Section 3.3 can be presented in a more general, categorical manner, which is described in *loc. cit.* In fact our so called union presheaf $\cup_r S_r$ is better described as a colimit (itself a generalization of coproduct) of the diagram

$$\ldots S_{r+1} \longrightarrow S_r \longrightarrow \ldots \longrightarrow S_1 \longrightarrow S_0$$

... but this is another story.

Acknowledgements

Thanks to Simon Ambler for useful discussions. I should emphasize again that these notes owe much to the ideas presented in Fiore, Plotkin and Turi's paper [9], and, more indirectly, to the recent work of Gabbay, Hofmann, and Pitts.

5 Appendix

5.1 Lists

We require the notion of a *finite list.* For the purposes of these notes, a (finite) **list** over a set S is an element of the set

$$[\,S\,] \overset{\text{def}}{=} \bigcup_{n<\omega} S^n$$

where S^n is the set of n-tuples of S and $S^0 \overset{\text{def}}{=} \{\,\epsilon\,\}$ where ϵ denotes the empty list. We denote a typical non-empty element of $[\,S\,]$ by $[\,s_1,\dots,s_n\,]$ or sometimes just $s_1,\dots,s_n$, and we write $s \in L$ to indicate that s occurs in the list (tuple) L. We write $\mathsf{len}(l)$ for the length of any list l.

5.2 Abstract Syntax Trees

We adopt the following notation for finite trees: If T_1, T_2, T_3 and so on to T_n is a (finite) sequence of finite trees, then we write $\mathsf{C}\ T_1\ T_2\ T_3\ \dots\ T_n$ for the finite tree which has the form

$$\begin{tikzcd}
 & & & \mathsf{C} \arrow[dlll] \arrow[dll] \arrow[dl] \arrow[dr] & \\
T_1 & T_2 & T_3 & \dots & T_n
\end{tikzcd}$$

Each T_i is itself of the form $\mathsf{C}'\ T_1'\ T_2'\ T_3'\ \dots\ T_m'$. We call C a **constructor** and say that C takes n arguments. Any constructor which takes 0 arguments is a **leaf** node. We call C the **root** node of the tree. The roots of the trees T_i are called the **children** of C. The constructors are **labels** for the *nodes* of the tree. Each of the T_i above is a **subtree** of the whole tree—in particular, any leaf node is a subtree.

If we say that A is a constructor which takes two arguments, and S and V constructors which takes one argument, then the tree in Figure 6 is denoted by $\mathsf{A}\ (\mathsf{V}\ v^2)\ (\mathsf{A}\ (\mathsf{V}\ v^2)\ \mathsf{S}\ (\mathsf{V}\ v^8))$. Note that in this (finite) tree, we regard each node as a constructor. To do this, we can think of any v^i as constructors which take no arguments!!. These form the *leaves* of the tree. We call the root of the tree the **outermost** constructor, and refer to trees of this kind as **abstract syntax** trees. We often refer to an abstract syntax tree by its outermost constructor—the tree above is an "A" expression.

5.3 Inductively Defined Sets

In this section we introduce a method for defining sets. Any such set will be known as an *inductively defined set.* Let us first introduce some notation. We

$$\begin{tikzcd}
 & \mathsf{A} \arrow[dl] \arrow[dr] & & \\
\mathsf{V} \arrow[d] & & \mathsf{A} \arrow[dl] \arrow[dr] & \\
v^2 & \mathsf{V} \arrow[d] & & \mathsf{S} \arrow[d] \\
 & v^3 & & \mathsf{V} \arrow[d] \\
 & & & v^8
\end{tikzcd}$$

Fig. 6. An Abstract Syntax Tree

let U be any set. A **rule** R is a pair (H, c) where $H \subseteq U$ is any finite set, and $c \in U$ is any element. Note that H might be $\varnothing$, in which case we say that R is a **base** rule. Sometimes we refer to base rules as **axioms**. If H is non-empty we say R is an **inductive** rule. In the case that H is non-empty we might write $H = \{ h_1, \ldots, h_k \}$ where $1 \leq k$. We can write down a base rule $R = (\varnothing, c)$ using the following notation

Base

$$\frac{}{c}\,(R)$$

and an inductive rule $R = (H, c) = (\{ h_1, \ldots, h_k \}, c)$ as

Inductive

$$\frac{h_1 \quad h_2 \quad \ldots \quad h_k}{c}\,(R)$$

Given a set U and a set $\mathcal{R}$ of rules based on U, a **deduction** is a finite tree with nodes labelled by elements of U such that

• each leaf node label c arises as a base rule $(\varnothing, c) \in \mathcal{R}$
• for any non-leaf node label c, if H is the set of children of c then $(H, c) \in \mathcal{R}$ is an inductive rule.

We then say that the set **inductively defined** by $\mathcal{R}$ consists of those elements $u \in U$ which have a deduction with root node labelled by u.

Example 1. 1. Let U be the set $\{ u_1, u_2, u_3, u_4, u_5, u_6 \}$ and let $\mathcal{R}$ be the set of rules

$$\{ R_1 = (\varnothing, u_1), R_2 = (\varnothing, u_3), R_3 = (\{ u_1, u_3 \}, u_4), R_4 = (\{ u_1, u_3, u_4 \}, u_5) \}$$

Then a deduction for u_5 is given by the tree

$$\begin{tikzcd}
 & u_5 \arrow[dl] \arrow[d] \arrow[dr] & & \\
u_1 & u_3 & u_4 \arrow[dl] \arrow[dr] & \\
 & u_1 & & u_3
\end{tikzcd}$$

which is more normally written up-side down and in the following style

$$\dfrac{\dfrac{}{u_1}\,R_1 \qquad \dfrac{}{u_3}\,R_2 \qquad \dfrac{\dfrac{}{u_1}\,R_1 \qquad \dfrac{}{u_3}\,R_2}{u_4}\,R_3}{u_5}\,R_4$$

2. A set $\mathcal{R}$ of rules for defining the set $E \subseteq \mathbb{N}$ of even numbers is $\mathcal{R} = \{ R_1, R_2 \}$ where

$$\frac{-}{0}\,(R_1) \qquad\qquad \frac{e}{e+2}\,(R_2)$$

Note that rule R_2 is, strictly speaking, a rule **schema**, that is e is acting as a variable. There is a "rule" for each instantiation of e. A deduction of 6 is given by

$$\dfrac{\dfrac{\dfrac{\dfrac{-}{0}\,(R_1)}{0+2}\,(R_2)}{2+2}\,(R_2)}{4+2}\,(R_2)$$

3. Let V be a set of **propositional variables**. The set of (first order) propositions *Prop* is inductively defined by the rules below. There are two distinguished (atomic) propositions true and false. Each proposition denotes a finite tree. In fact true and false are constructors with zero arguments, as is each p. The remaining logical connectives are constructors with two arguments, and are written in a sugared (infix) notation.

$$\frac{-}{v}\,[v \in V] \qquad \frac{}{\mathsf{false}} \qquad \frac{}{\mathsf{true}} \qquad \frac{\phi \quad \psi}{\phi \wedge \psi} \qquad \frac{\phi \quad \psi}{\phi \vee \psi} \qquad \frac{\phi \quad \psi}{\phi \to \psi}$$

5.4 Rule Induction

In this section we see how inductive techniques of proof which the reader has met before fit into the framework of inductively defined sets. We write $\phi(x)$ to denote a proposition about x. For example, if $\phi(x) \stackrel{\text{def}}{=} x \geq 2$, then $\phi(3)$ is true and $\phi(0)$ is false. If $\phi(a)$ is *true* then we often say that $\phi(a)$ **holds**.

We present in fig. 7 a useful principle called **Rule Induction**. It will be used throughout the remainder of these notes.

Let I be inductively defined by a set of rules $\mathcal{R}$. Suppose we wish to show that a proposition $\phi(i)$ holds for all elements $i \in I$, that is, we wish to prove

$$\forall i \in I. \quad \boxed{\phi(i)}\,.$$

Then all we need to do is

– for every base rule $\frac{}{b} \in \mathcal{R}$ prove that $\phi(b)$ holds; and
– for every inductive rule $\frac{h_1 \ldots h}{c} \in \mathcal{R}$ prove that whenever $h_i \in I$,

$$(\phi(h_1) \wedge \phi(h_2) \wedge \ldots \wedge \phi(h_k)) \quad \Longrightarrow \quad \phi(c)$$

We call the propositions $\phi(h_j)$ **inductive hypotheses**. We refer to carrying out the bulleted (•) tasks as "verifying **property closure**".

Fig. 7. Rule Induction

The Principle of Mathematical Induction arises as a special case of Rule Induction. We can regard the set $\mathbb{N}$ as inductively defined by the rules

$$\frac{}{0}\,(\mathit{zero}) \qquad \frac{n}{n+1}\,(\mathit{add1})$$

Suppose we wish to show that $\phi(n)$ holds for all $n \in \mathbb{N}$, that is $\forall n \in \mathbb{N}.\phi(n)$. According to Rule Induction, we need to verify

• property closure for *zero*, that is $\phi(0)$; and
• property closure for *add1*, that is for every natural number n, $\phi(n)$ implies $\phi(n+1)$, that is $\forall n \in \mathbb{N}.\ (\phi(n) \implies \phi(n+1))$

and this amounts to precisely what one needs to verify for Mathematical Induction.

1. Here is another example of abstract syntax trees defined inductively. Let a set of constructors be $\mathbb{Z} \cup \{+, -\}$. The integers will label leaf nodes, and $+$, $-$ will take two arguments written with an infix notation. The set of abstract syntax trees $\mathcal{T}$ inductively defined by these constructors is given by

$$\frac{}{n} \qquad \frac{T_1 \quad T_2}{T_1 + T_2} \qquad \frac{T_1 \quad T_2}{T_1 - T_2}$$

Note that the base rules correspond to leaf nodes. In the example tree

$$\begin{tikzcd}
 & & + \arrow[dl] \arrow[dr] & \\
 & - \arrow[dl] \arrow[dr] & & 2 \\
55 & & 7 &
\end{tikzcd}$$

$55 - 7$ is a subtree of $(55 - 7) + 2$, as are the leaves 55, 7 and 2.
The principle of **structural induction** is defined to be an instance of rule induction when the inductive definition is of abstract syntax trees. Make sure you understand that if $\mathcal{T}$ is an inductively defined set of syntax trees, to prove $\forall T \in \mathcal{T}.\phi(T)$ we have to prove:

- $\phi(L)$ for each leaf node L; and
- assuming $\phi(T_1)$ and ... and $\phi(T_n)$ prove $\phi(C(T_1, \ldots, T_n))$ for *each* constructor C and *all* trees $T_i \in \mathcal{T}$.

These two points are precisely property closure for base and inductive rules.
Consider the proposition $\phi(T)$ given by $L(T) = N(T) + 1$ where $L(T)$ is the number of leaves in T, and $N(T)$ is the number of $+, -$-nodes of T. We can prove by structural induction

$$\forall T \in \mathcal{T}. \quad \boxed{L(T) = N(T) + 1}$$

where the functions $L, N \colon \mathcal{T} \to \mathbb{N}$ are defined recursively by

- $L(n) = 1$ and $L(+(T_1, T_2)) = L(T_1) + L(T_2)$ and $L(-(T_1, T_2)) = L(T_1) + L(T_2)$
- $N(n) = 0$ and $N(+(T_1, T_2)) = N(T_1) + N(T_2) + 1$ and $N(-(T_1, T_2)) = N(T_1) + N(T_2) + 1$

This is left as an *exercise.*

Sometimes it is convenient to add a rule R to a set $\mathcal{R}$, which does not alter the resulting set I. We say that a rule

$$\frac{h_1 \quad \ldots \quad h_k}{c}\, R$$

is a **derived** rule of $\mathcal{R}$ if there is a deduction tree whose leaves are either the conclusions of base rules or are instances of the h_i, and the conclusion is c. The rule R is called **admissible** if one can prove

$$(h_1 \in I) \wedge \ldots \wedge (h_k \in I) \implies (c \in I)$$

Proposition 1. *Let I be inductively defined by $\mathcal{R}$, and suppose that R is a derived rule. Then the set I' inductively defined by $\mathcal{R} \cup \{ R \}$ is also I. Any derived rule is admissible.*

Proof. It is clear that $I \subset I'$. It is an exercise in rule induction to prove that $I' \subset I$. Verify property closure for each of the rules in $\mathcal{R} \cup \{ R \}$, the property $\phi(i) \stackrel{\text{def}}{=} i \in I$. It is clear that derived rules are admissible.

5.5 Recursively Defined Functions

Let I be inductively defined by a set of rules $\mathcal{R}$, and A any set. A function $f\colon I \to A$ can be defined by

• specifying an element $f(b) \in A$ for every base rule $\frac{}{b} \in \mathcal{R}$; and
• specifying $f(c) \in A$ in terms of $f(h_1) \in A$ and $f(h_2) \in A$ .... and $f(h_k) \in A$ for every inductive rule $\frac{h_1 \ldots, h}{c} \in \mathcal{R}$,

provided that each instance of a rule in $\mathcal{R}$ introduces a different element of I—why do we need this condition? When a function is defined in this way, it is said to be **recursively** defined.

Example 2. 1. The factorial function $F\colon \mathbb{N} \to \mathbb{N}$ is usually defined recursively. We set

• $F(0) \stackrel{\text{def}}{=} 1$ and
• $\forall n \in \mathbb{N}.F(n+1) \stackrel{\text{def}}{=} (n+1) * F(n)$.

Thus $F(3) = (2+1)*F(2) = 3*2*F(1) = 3*2*1*F(0) = 3*2*1*1 = 6$. Are there are brackets missing from the previous calculation? If so, insert them.

2. Consider the propositions defined on page 172. Suppose that ψ_i and x_i are propositions and propositional variables for $1 \leq i \leq n$. Then there is a recursively defined function *Prop* $\to$ *Prop* whose action is written $\phi \mapsto \phi\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\}$ which computes the *simultaneous substitution* of the ϕ_i for the x_i where the x_i *are distinct.* We set

• $x\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\} \stackrel{\text{def}}{=} \psi_j$ if x is x_j;
• $x\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\} \stackrel{\text{def}}{=} x$ if x is none of the x_i;
•

$$\begin{aligned}&(\phi \wedge \phi')\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\} \stackrel{\text{def}}{=} \\ &\qquad (\phi\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\}) \wedge (\phi'\{\psi_1, \ldots, \psi_n/x_1, \ldots, x_n\})\end{aligned}$$

• The other clauses are similar.

References

1. S. J. Ambler and R. L. Crole and A. Momigliano. Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction. Accepted for the 15th International Conference on Theorem Proving in Higher Order Logics, 20-23 August, Virginia, U.S.A. Springer Verlag Lecture Notes in Computer Science 2410, 2002.
2. R. Backhouse and R. L. Crole and J. Gibbons. Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. Revised lectures from an International Summer School and Workshop, Oxford, UK, April 2000. Springer Verlag Lecture Notes in Computer Science 2297, 2002.
3. M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer-Verlag, 1985.
4. M. Barr and C. Wells. *Category Theory for Computing Science.* International Series in Computer Science. Prentice Hall, 1990.

5. R. L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993. xvii+335 pages, ISBN 0521450926HB, 0521457017PB.
6. N. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
7. J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, Apr. 1995. Springer-Verlag LNCS 902.
8. J. Despeyroux, F. Pfenning, and C. Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, Apr. 1997. Springer-Verlag LNCS.
9. M. Fiore and G. D. Plotkin and D. Turi. Abstract Syntax and Variable Binding. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 193–202, Trento, Italy, 1999. IEEE Computer Society Press.
10. P.J. Freyd and A. Scedrov. *Categories, Allegories.* Elsevier Science Publishers, 1990. Appears as Volume 39 of the North-Holland Mathematical Library.
11. M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, 1999. IEEE Computer Society Press.
12. A. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In J.J. Joyce and C.-J.H. Seger, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 414–427, Vancouver, Canada, Aug. 1993. University of British Columbia, Springer-Verlag, published 1994.
13. A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190, Turku, Finland, August 1996. Springer-Verlag.
14. M. Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.
15. F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on systems in higher-order abstract syntax. In *Proc. ICALP'01*, number 2076 in LNCS, pages 963–978. Springer-Verlag, 2001.
16. Bart Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Elsevier, 1999.
17. P.T. Johnstone. *Topos Theory.* Academic Press, 1977.
18. J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.
19. S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.
20. Saunders Mac Lane and Ieke Moerdijk. Topos theory. In M. Hazewinkel, editor, *Handbook of Algebra, Vol. 1*, pages 501–528. North-Holland, Amsterdam, 1996.
21. Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Universitext. Springer-Verlag, New York, 1992.

22. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transaction in Computational Logic*, 2001. To appear.
23. J. McKinna and R. Pollack. Some Type Theory and Lambda Calculus Formalised. To appear in Journal of Automated Reasoning, Special Issue on Formalised Mathematical Theories (F. Pfenning, Ed.),
24. C. McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides*. Oxford University Press, 1991.
25. F. Pfenning. Computation and deduction. Lecture notes, 277 pp. Revised 1994, 1996, to be published by Cambridge University Press.
26. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.
27. B.C. Pierce. *Basic Category Theory for Computer Scientists.* Foundations of Computing Series. The MIT Press, 1991.
28. A. M. Pitts. Nominal logic: A first order theory of names and binding. In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001, Sendai, Japan, October 29-31, 2001, Proceedings*, volume 2215 of *Lecture Notes in Computer Science*, pages 219–242. Springer-Verlag, Berlin, 2001.
29. A. Poigné. Basic category theory. In *Handbook of Logic in Computer Science, Volume 1.* Oxford University Press, 1992.
30. M.B. Smyth and G.D. Plotkin. The Category Theoretic Solution of Recursive Domain Equations. In *SIAM Journal of Computing*, 1982, volume 11(4), pages 761–783.
31. Paul Taylor. *Practical Foundations of Mathematics.* Number 59 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1999.
32. R. F. C. Walters. *Categories and Computer Science.* Number 28 in Cambridge Computer Science Texts. Cambridge University Press, 1991.

A Mathematical Semantics for Architectural Connectors

J.L. Fiadeiro[1], A. Lopes[2], and M. Wermelinger[3]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt

[3] Dep. of Informatics, Faculty of Sciences and Technology, New University of Lisbon
Quinta da Torre, 2829-516 Caparica, Portugal
mw@di.fct.unl.pt

Abstract. A mathematical semantics is proposed for the notion of architectural connector, in the style defined by Allen and Garlan, that builds on Goguen's categorical approach to General Systems Theory and other algebraic approaches to specification, concurrency, and parallel program design. This semantics is, essentially, ADL-independent, setting up criteria against which formalisms can be evaluated according to the support that they provide for architectural design. In particular, it clarifies the role that the separation between computation and coordination plays in supporting architecture-driven approaches to software construction and evolution. It also leads to useful generalisations of the notion of connector, namely through the use of multiple formalisms in the definition of the glue and the roles, and their instantiations with programs or system components that can be implemented in different languages or correspond to "real-world" components.

1 Introduction

Architectural connectors have emerged as a powerful tool for supporting the description of the overall organisation of systems in terms of components and their interactions [6,27]. According to [1], an architectural connector (type) can be defined by a set of *roles* and a *glue* specification. For instance, a typical client-server architecture can be captured by a connector type with two roles – client and server – which describe the expected behaviour of clients and servers, and a glue that describes how the activities of the roles are coordinated (e.g. asynchronous communication between the client and the server). The roles of a connector type can be *instantiated* with specific components of the system under construction, which leads to an overall system structure consisting of components and connector instances establishing the interactions between the components.

The similarities between architectural constructions as informally described above and parameterised programming [17] are rather striking and have been more recently developed in [18]. The view of architectures that is captured by the principles and formalisms used in parameterised programming is reminiscent of Module Interconnection Languages and Interface Definition Languages. This perspective is somewhat different from the one followed in the work of Allen, Garlan and other researchers in Software Architectures. Software Architectures in this more recent sense focus instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation. As explained in [1], this kind of organisation is founded on *interaction* in the behavioural sense, which explains why process calculi are preferred to the functional flavour of equational specifications.

In [10,12], we showed that the mathematical "technology" of parameterisation can also be used for the formalisation of architectural connectors in the interaction sense. More concretely, we used preliminary work on an algebraic approach to parallel program design [11], in the tradition of the categorical approach to General Systems Theory also developed by Goguen [16], in order to bring together architectural principles and the categorical approach to system specification and design. In this paper, we provide a more thorough account of the application of categorical principles to architectural design by bringing to bear the developments on an algebraic semantics for the coordination paradigm as expose in [13], techniques for associating mobility with architectural connectors [29], and the beginnings of a calculus for composing architectural connectors [30].

The mathematical framework that we propose for formalising architectural principles is not specific to any particular Architecture Description Language (ADL). In fact, it will emerge from the examples that we shall provide that, contrarily to most other formalisations of Software Architecture concepts that we have seen, Category Theory is not another semantic domain for the formalisation of the description of components and connectors (like, say, the use of CSP in [1] or first-order logic in [26]). Instead, it provides for the very semantics of "interconnection", "configuration", "instantiation" and "composition", i.e. the principles and design mechanisms that are related to the gross modularisation of complex systems. Category Theory does this at a very abstract level because what it proposes is a toolbox that can be applied to whatever formalism is chosen for modelling the behaviour of systems as long as that formalism satisfies some structural properties. It is precisely the structural properties that make a formalism suitable for supporting architectural design that we shall make our primary focus.

Having this goal in mind, we start, in sections 2 and 3, by presenting a specific category corresponding to a specific program design language, and the way architectural connectors can be formalised over that category. This language – CommUnity – was developed precisely as a vehicle for illustrating the categorical approach to system design in general, and the role of architectures in this process in particular. Although it borrows principles from other languages that can be found in the literature, it takes to an extreme the separation between the support that it provides for the computations that are performed locally by components, and the mechanisms that are made available for coordinating the joint behaviour of components in a system. Hence, the purpose of introducing CommUnity at some length is to distil what we have found to be the minimal features that make up an architecture description language. In a companion paper [3], we show how these features have found their way into a full fledged object-oriented modelling language.

We proceed in section 4 by abstracting from CommUnity the distilled categorical properties that make an *architecture school.* We then show how the proposed mathematical framework leads to useful generalisations of the notion of architectural connector. We show how to support the use of different formalisms for describing the roles and the glue of a connector, namely a declarative formalism for the roles and a procedural one for the glue. We show how refinement mechanisms can interact with the application of architectural connectors as a means of supporting stepwise construction of systems. By this we mean both the possibility of establishing an architecture at the earlier phases of design that can then be refined by replacing abstract descriptions of the components involved by programs, and the support for a compositional evolution of the system through the addition, substitution or deletion of components or connectors. Furthermore, we show how we can support heterogeneity of components as a means of supporting the reuse of legacy components and the description of systems that incorporate non-software components.

Finally, in section 5, we show how to support several operations on connectors, giving the software architecture the ability to reuse connectors in the construction of the connectors that apply to a specific style.

The style of the paper is not purely "mathematical". It includes mathematical definitions of the main concepts, but we tried to make appeal more to the intuition than to rely on formality. Familiarity with basic notions and constructions of Category Theory is useful but there should be enough examples, and explanations around them, for readers acquainted with Software Architectures to make their way without much pain.

2 System Configuration in CommUnity

To illustrate the categorical approach that we wish to put forward for formalising architectural connectors and architectural descriptions, we will use the program design language CommUnity. CommUnity is a language similar to Unity [8] and Interacting Processes [14] that was initially developed to show how programs fit into Goguen's categorical approach to General Systems Theory [16]. Since then, the language and the design framework have been extended to provide a formal platform for architectural design of open, reactive, reconfigurable systems.

2.1 Component Design

One of the extensions that we have made to CommUnity since its original definition in [11] concerns the support for higher levels of design. At such levels of design, the architecture of the system is given in terms of components that are not necessarily programs but abstractions of programs – called *designs* – that can be *refined* into programs in later stages of the development process. Furthermore, some of these designs may account for components of the real-world with which the software components will be interconnected. Typically, such abstractions derive from requirements that have been specified in some logic or other mathematical models of the behaviour of real-world components. In this paper, we shall not address the process of deriving designs from specifications or other models. We will concentrate on the mathematical support that can be given to the process of building an architecture of the system from given designs.

The goal of supporting abstraction is not only to support a stepwise approach to software *construction*, but also an architectural design layer that is closer to the application domain and, hence, can be used for driving the evolution of the system according to the changes that occur in the domain. An important part of this evolution may consist of changes in the nature of components, with real-world components being replaced or controlled by software components, or software components being reprogrammed in another language, which again stresses the importance of supporting abstraction in architectural design.

The support for abstraction in CommUnity is twofold. On the one hand, designs account for what is usually called *underspecification*, i.e. they are structures that do not denote unique programs but collections of programs. On the other hand, designs can be defined over a collection of data types that do not correspond necessarily to those that will be available in the final implementation platform. Therefore, there are two refinement procedures that have to be accounted for in CommUnity. On the one hand, the removal of underspecification from designs in order to define programs over the layer of abstraction defined by the data types that have been used. On the other hand, the reification of the data types in order to bring programs into the target implementation environment.

In order to address these two processes in reasonable depth, we would need more than one paper! The choice of data types determines, essentially, the nature of the elementary computations that can be performed locally by the components, which are abstracted as operations on data elements. Although such elementary computations also determine the granularity of the services that components can provide and, hence, the granularity of the interconnections that can be established at a given layer of abstraction, data refinement is more concerned with the computational aspects of systems than with coordination. Because architectural design is more concerned with the coordination of system behaviour within a given layer of abstraction, we decided to give more emphasis to refinement of designs for a fixed choice of data types and omit any discussion on data refinement.

Given this, we shall assume a collection of data types to be fixed. In order to remain independent of any specific language for the definition of these data types, we take them in the form of a first-order algebraic specification. That is to say, we assume a data signature $<S,\Omega>$, where S is a set (of sorts) and Ω is a $S^{*}\times S$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations.

A CommUnity component design over such a data type specification is of the form

```
design  P is
out     out(V)
in      in(V)
prv     prv(V)
do       []                    g[D(g)]: L(g), U(g) →  R(g)
       g sh(Γ)
         []           prv      g[D(g)]: L(g), U(g) →  R(g)
       g prv(Γ)
```

where

- V is the set of *channels*. Channels can be declared as *input*, *output* or *private*. Input channels are used for reading data from the environment of the component. The component has no control on the values that are made available in such channels. Moreover, reading a value from an input channel does not "consume" it: the value remains available until the environment decides to replace it. Output and private channels are controlled locally by the component, i.e. the values that, at any given moment, are available on these channels cannot be modified by the environment. Output channels allow the environment to read data produced by the component. Private channels support internal activity that does not involve the environment in any way. We use $loc(V)$ to denote the union $prv(V) \cup out(V)$, i.e. the set of local channels. Each channel v is typed with a sort $sort(v) \in S$.
- by Γ we denote the set of *action names*. The named actions can be declared either as *private* or *shared* (for simplicity, we only declare which actions are private). Private actions represent internal computations in the sense that their execution is uniquely under the control of the component. Shared actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. The significance of naming actions will become obvious below; the idea is to provide points of *rendez-vous* at which components can synchronise.
- For each action name g, the following attributes are defined:
  - $D(g)$ is a subset of $loc(V)$ consisting of the local channels that can be effected by executions of the action named by g. This is what is sometimes called the *write frame* of g. For simplicity, we will omit the explicit reference to the write frame when $R(g)$ is a conditional multiple assignment (see below), in which case $D(g)$ can be inferred from the assignments. Given a local channel v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$.
  - $L(g)$ and $U(g)$ are two conditions such that $U(g) \supset L(g)$. These conditions establish an interval in which the enabling condition of any guarded command that implements g must lie. The condition $L(g)$ is a lower bound for enabledness in the sense that it is implied by the enabling condition. Therefore, its negation establishes a *blocking* condition. On the other hand, $U(g)$ is an upper bound in the sense that it implies the enabling condition, therefore establishing a *progress* condition. Hence, the enabling condition is fully determined only if $L(g)$ and $U(g)$ are equivalent, in which case we write only one condition.
  - $R(g)$ is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local channels from the write frame of g. As usual, these primed channels account for references to the values that the channels take after the execution of the action. These conditions are usually a conjunction of implications of the form $pre \supset pos$ where pre does not involve primed channels. They correspond to pre/post-condition specifications in the sense of Hoare. When $R(g)$ is such that the primed version of each local channel in the write frame of g is fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame $D(g)$ is empty, $R(g)$ is tautological, which we denote by *skip*.

Notice that CommUnity supports several mechanisms for underspecification – actions may be underspecified in the sense that their enabling conditions may not be fully determined (subject to refinement by reducing the interval established by *L* and *U*) and their effects on the channels may also not be fully determined.

When, for every $g \in \Gamma$, $L(g)$ and $U(g)$ coincide, and the relation $R(g)$ defines a conditional multiple assignment then the design is called a *program*. Notice that a program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in the context of a configuration in which these inputs have been instantiated with (local) channels of other components. The notion of configuration, and the execution of an open program in a given configuration, will be discussed further below. The behaviour of a closed program is as follows. At each execution step, any of the actions whose enabling condition holds of the current state can be executed, in which case its assignments are performed atomically. Furthermore, private actions that are infinitely often enabled are guaranteed to be selected infinitely often. See [22] for a model-theoretic semantics of CommUnity.

Designs can be parameterised by data elements (sorts and operations) indicated after the name of the component (see an example below). These parameters are instantiated at configuration time, i.e. when a specific component needs to be included in the configuration of the system being built, or as part of the reconfiguration of an existing system.

As an example, consider the following parameterised design.

```
design buffer [t:sort, bound:nat] is
in     i:t
out    o:t
prv    rd: bool, b: list(t)
do     put: |b|<bound→ b:=b.i
[] prv next: |b|>0∧¬rd →o:=head(b)||b:=tail(b)||rd:=true
[]     get: rd →rd:=false
```

This design is actually a (parameterised) program and the traditional notation of guarded commands was used accordingly. Notice in particular that the reference to the write frame of the actions was omitted: it can be inferred from the multiple assignments that they perform. This program models a buffer with a limited capacity and a FIFO discipline. It can store, through the action *put*, messages of sort *t* received from the environment through the input channel *i*, as long as there is space for them. The buffer can also discard stored messages, making them available to the environment through the output channel *o* and the action *next*. Naturally, this activity is possible only when there are messages in store and the current message in *o* has already been read by the environment (which is modelled by the action *get* and the private channel *rd*).

In order to illustrate the ability of CommUnity to support higher-level component design, we present below the design of a typical sender of messages. In this description, we are primarily concerned with the interaction between the sender and its environment, ignoring details of internal computations such as the production of messages.

```
design sender[t:sort] is
out    o:t
prv    rd: bool
do     prod[o,rd]: ¬rd,false→rd'
[]     send[rd]: rd,false→¬rd'
```

Notice that a *sender* cannot produce another message before the previous one has been processed. After producing a message, a sender expects an acknowledgement (modelled through the execution of *send*) to produce a new message.

In order to leave unspecified when and how many messages a *sender* will send and in which situations it will produce a new message, the progress conditions of *prod* and *send* are false (recall that the progress condition defines the upper bound for enabledness). Furthermore, the discipline of production was also left completely unspecified: the action *prod* includes the channel *o* in its write frame but the design does not commit to any specific way of changing the value of this channel.

From a mathematical point of view, (instantiated) CommUnity designs are structures defined as follows.

A signature is a tuple $<V,\Gamma,tv,ta,D>$ *where*

- V is an S-indexed family of mutually disjoint finite sets,
- Γ is a finite set,
- tv: $V\rightarrow\{out,in,prv\}$ is a total function,
- ta: $\Gamma\rightarrow\{sh,prv\}$ is a total function,
- D: $\Gamma\rightarrow 2^{loc(V)}$ is a total function.

A design is a pair $<\theta,\Delta>$ *where* $\theta=<V,\Gamma,tv,ta,D>$ *is a signature and* Δ*, the body of the design, is a tuple* $<R,L,U>$ *where:*

- R assigns to every action $g\in\Gamma$, a proposition over $V\cup D(g)'$,
- L and U assign a proposition over V to every action $g\in\Gamma$.

The reader who is familiar with parallel program design languages or earlier versions of CommUnity will have probably noticed the absence of initialisation conditions. The reason they were not included in CommUnity designs is because they are part of the configuration language of CommUnity, not the parallel program design language. That is to say, we take initialisation conditions as part of the mechanisms that relate to the building and management of configurations out of designs, not of the construction of designs themselves.

2.2 Configurations

So far, we have presented the primitives for the design of individual components, which are another variation on guarded commands. The features of these designs that are not shared with other parallel program design languages are those that concern design "in the large", i.e. the ability to design large systems from simpler components.

The model of interaction between components in CommUnity is based on action synchronisation and the interconnection of input channels of a component with output channels of other components. These are standard means of interconnecting software components. What distinguishes CommUnity from other parallel program design languages is the fact that such interactions between components are required to be made explicit by providing the corresponding name bindings. Indeed, parallel program design languages normally leave such interactions implicit by relying on the use of the same names in different components. In CommUnity, names are local to designs. This means that the use of the same name in different designs is treated as being purely accidental, and, hence, expresses no relationship between the components.

In CommUnity, name bindings are established as relationships between the signatures of the corresponding components, matching channels and actions of these components. These bindings are made explicit in configurations. A configuration determines a diagram containing nodes labelled with the signatures of the components that are part of the configuration. Name bindings are represented as additional nodes labelled with sets representing the actual interactions, and edges labelled with the projections that map each interaction to the corresponding component signatures.

For instance, a configuration in which the messages from a *sender* component are sent (to a receiver) through a bounded buffer defines the following diagram:

$$\begin{tikzcd} & \texttt{cable} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\substack{\texttt{o}\leftarrow\bullet\rightarrow\texttt{i} \\ \texttt{send}\rightarrow\bullet\leftarrow\texttt{put}}"] & \\ \texttt{signature(}\textbf{sender}\texttt{)} & & \texttt{signature(}\textbf{buffer}\texttt{)} \end{tikzcd}$$

The node labelled *cable* is the representation of the set of bindings. Because, as we have seen, channels and action names are typed and classified in different categories, not every pair of names is a valid binding. To express the rules that determine valid bindings, it is convenient to structure *cable* as a signature itself. Hence, in the case above, *cable* consists of an input channel to model the medium through which data is to be transmitted between the sender and the buffer, and a shared action for the two components to synchronise in order to transmit the data. Because, as we have already mentioned, names in CommUnity are local, the identities of the shared input channel and the shared action in *cable* are not relevant: they are just placeholders for the projections to define the relevant bindings. Hence, we normally do not bother to give them explicit names, and represent them through the symbol •.

The bindings themselves are established through the labels of the edges of the diagram. In the case above, the input channel of *cable* is mapped to the output channel *o* of *sender* and to the input channel *i* of *buffer*. This establishes an i/o-interconnection between *sender* and *buffer*. On the other hand, the actions *send* of *sender* and *put* of *buffer* are mapped to the shared action of *cable*. This defines that *sender* and *buffer* must synchronise each time either of them wants to perform the corresponding action. The fact that the mappings on action names and on channels go in opposite directions will be discussed below.

The arrows that we are using to define interconnections between components are also mathematical objects: they are examples of signature morphisms.

A morphism $\sigma:\theta_1\rightarrow\theta_2$ between signatures $\theta_1=<V_1,\Gamma_1,tv_1,ta_1,D_1>$ and $\theta_2=<V_2,\Gamma_2,tv_2,ta_2,D_2>$ is a pair $<\sigma_{ch},\sigma_{ac}>$ where

- σ_{ch}: $V_1\rightarrow V_2$ is a total function satisfying:
  - $sort_2(\sigma_{ch}(v))=sort_1(v)$ for every $v\in V_1$
  - $\sigma_{ch}(o)\in out(V_2)$ for every $o\in out(V_1)$
  - $\sigma_{ch}(i)\in out(V_2)\cup in(V_2)$ for every $i\in in(V_1)$
  - $\sigma_{ch}(p)\in prv(V_2)$ for every $p\in prv(V_1)$

- *σ_{ac}: $\Gamma_2 \rightarrow \Gamma_1$ is a partial mapping satisfying for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:*
  - *if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$*
  - *if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$*
  - *$\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$*
  - *σ_{ac} is total on $D_2(\sigma_{ch}(v))$ and $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$ for every $v \in loc(V_1)$*

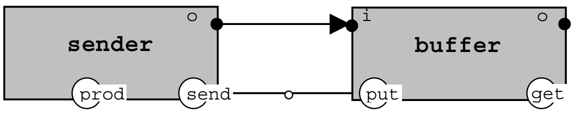
Signature morphisms represent more than the projections that arise from name bindings as illustrated above. A morphism σ from θ_1 to θ_2 is intended to support the identification of a way in which a component with signature θ_1 is embedded in a larger system with signature θ_2. This justifies the various constructions and constraints in the definition.

The function σ_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The fact that the two mappings go in opposite directions is justified as follows. Actions of the system constitute synchronisation sets of actions of the components. Because not every component is necessarily involved in every action of the system, the action mapping is partial. On the other hand, because each action of the component may participate in more than one synchronisation set, but each synchronisation set cannot induce internal synchronisations within the components, the relationship between the actions of the system and the actions of every component is functional from the former to the latter.

Input/output communication within the system is not modelled in the same way as action synchronisation. Synchronisation sets reflect parallel composition whereas with i/o-interconnections we wish to unify communication channels of the components. This means that, in the system, channels should be identified rather than paired. This is why mappings on channels and mappings on actions go in opposite directions. We will see that, as a result, the mathematical semantics of configuration diagrams induces fibred products of actions (synchronisation sets) and amalgamated sums of channels (equivalence classes of connected channels).

The other constraints are concerned with typing. Sorts of channels have to be preserved but, in terms of their classification, input channels of a component may become output channels of the system. This is because, in the absence of other constraints, the result of interconnecting an input channel of a component with an output channel of another component in the system is an output channel of the system. Mechanisms for internalising communication can be applied but they are not the default in a configuration. The last two conditions on write frames implies that actions of the system in which a component is not involved cannot have local channels of the component in its write frame. That is, change within a component is completely encapsulated in the structure of actions defined for the component.

The diagrams that we use for expressing configurations of interconnected components are also mathematical objects. Indeed, Category Theory goes a long way, as far as Mathematics in concerned, in providing a formal framework for configuration that is “graphical”. Even so, the notation can be simplified and made more user-friendly by adopting some features that are typical of languages for configurable distributed systems like [25]. For instance, the interconnection defined before can be described as follows.



The interconnection, i.e. the name bindings, are still represented explicitly but, instead of being depicted as a component, the cable is now represented, perhaps more intuitively, in terms of arcs that connect channels and actions directly. We shall not provide a full definition of the graphical language that we defined for CommUnity because it is self-explanatory. It is also easy to see that configurations in this notation are easily translated into categorical diagrams by transforming the interconnections into cables and morphisms.

So far, we have explained how interconnections between components can be established at the level of the signatures of their designs. It remains to explain how the corresponding designs are interconnected, i.e. what is the semantics of the configuration diagram once designs are taken into account. For that purpose, we need to extend the notion of morphism from signatures to designs.

A morphism $\sigma{:}P_1 \rightarrow P_2$ of designs $P_1 = <\theta_1, \Delta_1>$ and $P_2 = <\theta_2, \Delta_2>$, consists of a signature morphism $\sigma{:}\,\theta_1 \rightarrow \theta_2$ such that, for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:

1. $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$;
2. $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$;
3. $\Phi \vdash (U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g))))$;

*where Φ is the axiomatisation of the data type specification, $\vdash$ denotes validity in the first-order sense, and $\underline{\sigma}$ is the extension of σ to the language of expressions and conditions. Designs and their morphisms constitute a category **c-DSGN.***

A morphism $\sigma{:}P_1 \rightarrow P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that P_2 can be considered as having been obtained from P_1 through the superposition of additional behaviour, namely the interconnection of one or more components. The conditions on the actions require that the computations performed by the system reflect the interconnections established between its components. Condition 1 reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. This is because the other components in the system cannot interfere with the transformations that the actions of a given component make on its state, except possibly by removing some of the underspecification present in the component design.

Conditions 2 and 3 allow the bounds that the component design specifies for the enabling of the action to be strengthened but not weakened. Strengthening of the lower bound reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. On the other hand, it is clear that progress for a joint action can only be guaranteed when all the designs of the components involved can locally guarantee so.

The notion of morphism that we have just defined captures what in the literature on parallel program design is called "superposition" or "superimposition" [8,20]. See [11] for the categorical formalisation of different notions of superposition and their algebraic properties.

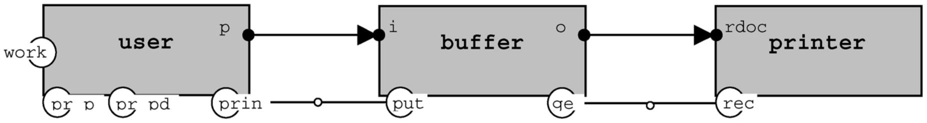
The semantics of configurations is given by a categorical construction: the colimit of the underlying diagrams. Taking the colimit of a diagram collapses the configuration into an object by internalising all the interconnections, thus delivering a design for the system as a whole. Furthermore, the colimit provides a morphism σ_i from each component design P_i in the configuration into the new design (that of the system). Each such morphism is essential for identifying the corresponding component within the system because the construction of the new design typically requires that the features of the components be renamed in order to account for the interconnections.

For instance, in the case of actions, the colimit represents every synchronisation set $\{g_1,\dots,g_n\}$ of actions of the components, as defined through the interconnections, by a single action $g_1\|\dots\|g_n$ whose occurrence captures the joint execution of the actions in the set. The transformations performed by the joint action are specified by the conjunction of the specifications of the local effects of each of the synchronised actions, i.e. $R(g_1\|\dots\|g_n)=\underline{\sigma}_1(R(g_1))\wedge\dots\wedge\underline{\sigma}_n(R(g_n))$ where the σ_i are the morphisms that connect the components to the system. The bounds on the guards of joint actions are also obtained through the conjunctions of the bounds specified by the components, i.e. $L(g_1\|\dots\|g_n)=\underline{\sigma}_1(L(g_1))\wedge\dots\wedge\underline{\sigma}_n(L(g_n))$ and $U(g_1\|\dots\|g_n)=\underline{\sigma}_1(U(g_1))\wedge\dots\wedge\underline{\sigma}_n(U(g_n))$.

At the level of the channels, instead of computing synchronisation sets, the colimit construction amalgamates the channels involved in each i/o-communication established by the configuration. Each such interaction is represented at the level of the system design as an output channel. From a mathematical point of view, these channels represent the quotient sets of channels defined by the equivalence relation that results from the i/o-interconnections.

Hence, colimits in CommUnity capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components. We can see this operation as a generalisation of the notion of superimposition as defined in [14].

An example of a more complex configuration is given below. It models the interconnection between a user and a printer via a buffer.



The user produces files that it stores in the private channel *w*. It can then convert them either to postscript or pdf formats, after which it makes them available for printing in the output channel *p*.

```
design user is
out    p:ps+pdf
prv    s: 0..2, w: MSWord
do     work[w,s]: s=0,false → s'=1
▯      pr_ps: s=1,false → p:=ps(w)||s:=2
▯      pr_pdf: s=1,false → p:=pdf(w)||s:=2
▯      print: s=2 → s:=0
```

The printer copies the files it downloads from the input channel *rdoc* into the private channel *pdoc*, after which it prints them.

```
design printer is
in     rdoc:ps+pdf
prv    busy: bool, pdoc: ps+pdf
do     rec: ¬busy → pdoc:=rdoc||busy:=true
□  prv end_print: busy → busy:=false
```

The configuration connects the user to the printer via a buffer as expected. The user "prints" by placing the file in the buffer: this is achieved through the synchronisation pair *{print,put}* and the i/o-interconnection *{p,i}*. The printer downloads from the buffer the files that it prints: this is achieved through the synchronisation pair *{get, rec}* and the i/o-interconnection *{o,rdoc}*.

The design of the system that results from the colimit of the configuration diagram contains two channels that account for the two i/o-interconnections *{p,i}* and *{o,rdoc}*, together with the private channels of the components. At the level of its actions, it generates the following shared actions (synchronisation sets):

- *{print,put}*, *{get, rec}* – these are required by the interconnections
- *{work}*, *{pr_ps}*, *{pr_pdf}*, *{work,get,rec}*, *{pr_ps,get,rec}*, *{pr_pdf,get,rec}* – these reflect the concurrent executions that respect the interconnections.

No other shared actions are possible because of the synchronisation requirements imposed on the components.

Because actions of the system are synchronisation sets of actions of the components, the evaluation of the guard of a chosen action can be performed in a distributed way by evaluating the guards of the component actions in the synchronisation set. The joint action will be executed iff all the local guards evaluate to *true*. The execution of the multiple assignment associated with the joint action can also be performed in a distributed way by executing each of the local assignments. What is important is that the atomicity of the execution must be guaranteed, i.e. the next system step should only start when all local executions have completed, and the i/o-communications should be implemented so that every local input channel is instantiated with the correct value – that which holds of the local state before any execution starts (synchronicity). Hence, the colimit of the configuration diagram should be seen as an abstraction of the actual distributed execution that is obtained by coordinating the local executions according to the interconnections, rather than the program that is going to be executed as a monolithic unit.

The fact that the computational part, i.e. the one that is concerned with the execution of the actions on the state, can be separated from the coordination aspects is, therefore, an essential property for guaranteeing that the operational semantics is compositional on the structure of the system as given through its configuration diagram. We will return to this aspect later on and provide a formal characterisation of what we mean by separating computation from coordination.

Not every diagram of designs reflects a meaningful configuration. For instance, it does not make sense to interconnect components by connecting two output channels. Indeed, we cannot guarantee that every diagram admits a colimit, meaning that there are diagrams that have no "semantics" as configurations.

The two following rules express the restrictions on the diagrams of designs that make them well-formed configurations.

- An output channel of a component cannot be connected with output channels of the same or other components;
- Private channels and private actions cannot be involved in the connections.

It is important to notice that the second rule establishes the configuration semantics of private actions and channels. It supports the intuitive semantics we gave in section 2.1, namely that private channels cannot be read by the environment and that the execution of shared actions is uniquely under the control of the component. Well-formed configurations are guaranteed to generate diagrams that admit colimits.

In Category Theory, the colimit construction applies to diagrams in which all the nodes are labelled with objects of the same category and all the edges are labelled with morphisms of the same category as well. However, in what concerns configurations, we have expressed interconnections at the level of the signatures of the component designs. This is because we have explained interconnections in terms of synchronisation of actions and i/o-communication, which are modelled at the level of signatures. How can we bring into a single category all the elements that are used in a configuration to make sure that the mathematics work?

In CommUnity, every signature θ defines a canonical design ***dsgn(θ)***: the one that is completely underspecified. More precisely, to each action *g* of the signature we assign *true* to *R(g)*, i.e. we make no commitments on the effects of the action on the channels, and *true* to *L(g)* and *true* to *U(g)*, i.e. we make no commitments as to the bounds of the enabling condition in the sense that we do not restrict the designs to which the cable can be mapped. It is trivial to check that, given a design *P* and a signature θ, every morphism $\sigma:\theta\rightarrow$***sig(P)*** is also a morphism of designs ***dsgn(θ)***$\rightarrow$*P*. Hence, we can replace every cable in a configuration diagram by the canonical design that it denotes, and take the colimit of the extended diagram as the semantics of the configuration. Knowing this, and for simplicity, we will keep using signatures in diagrams.

Although this solution satisfies our purposes in the sense that it provides the envisaged semantics for configurations, it also raises an interesting question. Are we loosing any expressive power by restricting interconnections to operate at the level of signatures? Would we obtain a more expressive configuration language by allowing interconnections to be expressed directly as diagrams in the category of designs, i.e. using any kind of designs as cables? After all, the application of Category Theory to General System Theory originally developed by Goguen does not distinguish between designs and their signatures…

This question is concerned with the degree of separation that CommUnity provides between computation and coordination, an issue that is central to Software Architectures. More precisely, the question is about the mechanisms that in CommUnity are relevant for coordinating the behaviour of components. If, indeed, extending configuration diagrams to full designs does not increase the interconnections that are allowed in the language, then coordination does not require the features that are absent from signatures, i.e. the computational part of actions does not interfere with coordination. If, on the contrary, we are able to show that we can establish more interconnections by using full designs as cables, then we will have proved that there is more to coordinating the interaction between components then synchronising actions and establishing i/o-communications.

It turns out that signatures provide, indeed, all that is necessary for interconnecting designs. To prove so, we have first to formulate the property at stake. Because this is a discussion that can be held independently of the language that is being used, leading to a mathematical formalisation of coordination, we postpone the formulation of the property and its proof to section 3.

3 Architectural Description in CommUnity

3.1 Architectural Connectors

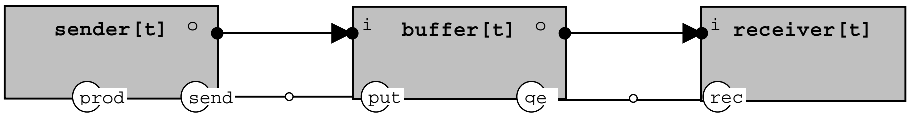
According to [1], an architectural connector (type) can be defined by a set of *roles*, that can be instantiated with specific components of the system under construction, and a *glue* specification that describes how the activities of the role instances are to be coordinated. Using the mechanisms that we have just described for configuration design in CommUnity, it is not difficult to come up with a formal notion of connector that has the same properties as those given in [1] for the language WRIGHT:

- *A connection consists of*
  - *two designs G and R, called the glue and the role of the connection, respectively;*
  - *a signature θ and two morphisms σ:**dsgn**$(\theta)\rightarrow G$,μ:**dsgn**$(\theta)\rightarrow R$ connecting the glue and the role.*
- *A connector is a finite set of connections with the same glue that, together, constitute a well-formed configuration.*

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \theta_i \arrow[u] \arrow[d] & \theta_n \arrow[ul] \arrow[d] \\
R_1 & R_i & R_n
\end{tikzcd}$$

- *The semantics of a connector is the colimit of the diagram formed by its connections.*

For instance, asynchronous communication through a bounded channel can be represented by a connector *Async* with two connections, as depicted below using the graphical notation that we have already introduced.



The glue of *Async* is the bounded buffer with FIFO discipline presented before. It prevents the *sender* from sending a new message when there is no space and prevents the *receiver* from reading a new message when there are no messages. The two roles – *sender* and *receiver* – define the behaviour required of the components to which the connector can be applied. For the *sender*, we require that no message be produced before the previous one has been processed. Its design is the one given already in section 2.1. For the *receiver*, we simply require that it has an action that models the reception of a message.

```
design receiver [t:sort] is
in i: t
do rec: true,false→skip
```

What we have described are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific designs. In WRIGHT [1], role instantiation has to obey a compatibility requirement expressed via the refinement relation of CSP. In CommUnity, the refinement relation is also defined over a category of designs but using a different notion of morphism.

Indeed, the notion of morphism defined in the previous section does not capture a refinement relation in the sense that it does not ensure that any implementation of the target provides an implementation for the source. For instance, it is easy to see that morphisms do not preserve the interval assigned to the guard of each action. Given that the aim of the defined morphisms was to capture the relationship that exists between systems and their components, this is hardly surprising. It is well known in languages such as CSP that the parallel composition of a collection of processes does not refine, necessarily, any of the individual processes.

The notion of morphism that captures the refinement relation in CommUnity is the following:

A refinement morphism $\sigma: P_1 \rightarrow P_2$ *of designs* $P_1 = <\theta_1, \Delta_1>$ *and* $P_2 = <\theta_2, \Delta_2>$ *is a pair* $<\sigma_{ch}, \sigma_{ac}>$ *where*

- $\sigma_{ch}: V_1 \rightarrow Term(V_2)$ *is a total function mapping the channels of* P_1 *to the class of terms built from the channels of* P_2 *and the data type operations. This mapping is required to satisfy, for every* $v \in V_1$, $o \in out(V_1)$, $i \in in(V_1)$, $p \in prv(V_1)$:
  - $sort_2(\sigma_{ch}(v)) = sort_1(v)$
  - $\sigma_{ch}(o) \in out(V_2)$
  - $\sigma_{ch}(i) \in in(V_2)$
  - $\sigma_{ch}(p) \in Term(loc(V_2))$
  - $\sigma_{ch} \downarrow (out(V_1) \cup in(V_1))$ *is injective*
- $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ *is a partial mapping satisfying for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined:*
  - *if* $g \in sh(\Gamma_2)$ *then* $\sigma_{ac}(g) \in sh(\Gamma_1)$
  - *if* $g \in prv(\Gamma_2)$ *then* $\sigma_{ac}(g) \in prv(\Gamma_1)$
  - *if* $g \in sh(\Gamma_1)$ *then* $\sigma_{ac}^{-1}(g) \neq \varnothing$
  - $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq \underline{D}_2(g)$
  - σ_{ac} *is total on* $\underline{D}_2(\sigma_{ch}(v))$ *and* $\sigma_{ac}(\underline{D}_2(\sigma_{ch}(v))) \subseteq D_1(v)$ *for every* $v \in loc(V_1)$

  and, furthermore,
- *for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined:*
  1. $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$;
  2. $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$;
- *for every* $g_1 \in \Gamma_1$,
  3. $\Phi \vdash (\underline{\sigma}(U_1(g_1)) \supset \underset{\sigma_{ac}(g_2)=g_1}{\Delta} U_2(g_2))$

where $\underline{D}$ *is the extension of D to the language of expressions. Designs and their refinement morphisms constitute a category* ***r-DSGN.***

A refinement morphism is intended to support the identification of a way in which a design P_1 (its source) is refined by a more concrete design P_2 (its target).

The function σ_{ch} identifies for each input (resp. output) channel of P_1 the corresponding input (resp. output) channel of P_2. Notice that, contrarily to what happens with the component-of relationship, refinement does not change the border between the system and its environment and, hence, input channels can no longer be mapped to output channels. Moreover, refinement morphisms allow each private channel of P_1 to be expressed in terms of the local channels of P_2 through an expression. The evaluation of such an expression may involve some computation as captured through the use of operations from the underlying data types. Naturally it is required that the sorts of channels be preserved.

The mapping σ_{ac} identifies for each action g of P_1, the set of actions of P_2 that implements g – given by $\sigma_{ac}^{-1}(g)$. This set is a menu of refinements for action g and can be empty for private actions. However, every action that models interaction between the component and its environment has to be implemented.

The actions for which σ_{ac} is left undefined (the new actions) and the channels which are not involved in $\sigma_{ch}(V_1)$ (the new channels) introduce more detail in the description of the component.

Conditions 2 and 3 require that the interval defined by the blocking and progress conditions of each action (in which the enabling condition of any guarded command that implements the action must lie) be preserved or reduced. This is intuitive because refinement, pointing in the direction of implementations, should reduce underspecification. This is also the reason why the effects of the actions of the more abstract design are required to be preserved or made more deterministic.

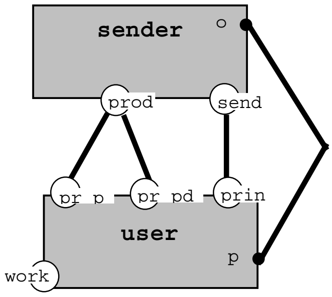
For instance, *sender* is refined by *user* via the refinement morphism η:*sender*→*user* defined by

$\eta_{ch}(o)=p,\ \eta_{ch}(rd)=(s=2)$

$\eta_{ac}(pr\_ps)=\eta_{ac}(pr\_pdf)=prod,\ \eta_{ac}(print)=send.$

In *user*, the production of messages (to be sent) is modelled by any of the actions *pr_ps* and *pr_pdf* and the messages are made available in the output channel *p*. Notice that the production of messages, that was left unspecified in *sender,* is completely defined in *user*: it corresponds to the conversion of the files stored in *w* to ps or pdf formats.

In the graphical notation, refinement is represented through bold (thick) lines:



Likewise, *printer* is a refinement of *receiver* via the refinement morphism κ:*receiver*→*printer* defined by

$\kappa_{ch}(i)=rdoc$
$\kappa_{ac}(rec)=rec$

In *printer*, the reception of a message from the input channel (named *rdoc)* corresponds to downloading it into the private channel *pdoc*. This action is only enabled if the previous message has already been printed.

An instantiation of a connector in CommUnity can now be defined as follows:

- *An instantiation of a connection with role R consists of a design P together with a refinement morphism ϕ:R→P.*
- *An instantiation of a connector consists of an instantiation for each of its connections.*

In order to define the semantics of such an instantiation, notice that each instantiation ϕ*:R→P* of a connection can be composed with μ to define $\mu;\phi:\theta\rightarrow$***sig****(P)*. This is because θ is according to the rules that define well-formed configurations and, hence, has no private channels, which means that the refinement morphism has the same properties as a composition morphism over θ. As we have already argued, every such signature morphism can be lifted to a design morphism $\mu;\phi$:***dsgn****(*θ*)→P*. Hence, an instantiation of a connector defines a diagram in ***c-DSGN*** that connects the role instances to the glue.

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d, "\mu_1"] \arrow[dd, bend right] & \theta_i \arrow[u] \arrow[d, "\mu_i"] \arrow[dd, bend right] & \theta_n \arrow[ul] \arrow[d, "\mu_n"] \arrow[dd, bend right] \\
R_1 \arrow[d, "\phi_1"] & R_i \arrow[d, "\phi_i"] & R_n \arrow[d, "\phi_n"] \\
P_1 & P_i & P_n
\end{tikzcd}$$

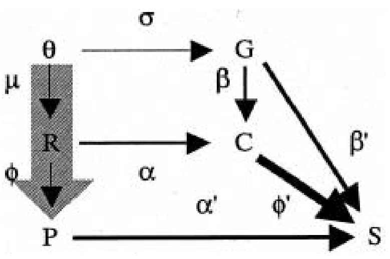
Moreover, because each connection is according to the rules set for well-formed configurations as detailed in the previous section, the diagram defined by the instantiation is, indeed, a configuration and, hence, has a colimit.

The semantics of a connector instantiation is the colimit of the diagram in ***c-DSGN*** *formed as described above by composing the role morphism of each connection with its instantiation.*

Because, as we have already argued, colimits in ***c-DSGN*** express parallel composition, this semantics agrees with the one provided in [1] for the language WRIGHT. In the next section, we shall take this analogy with WRIGHT one step further.

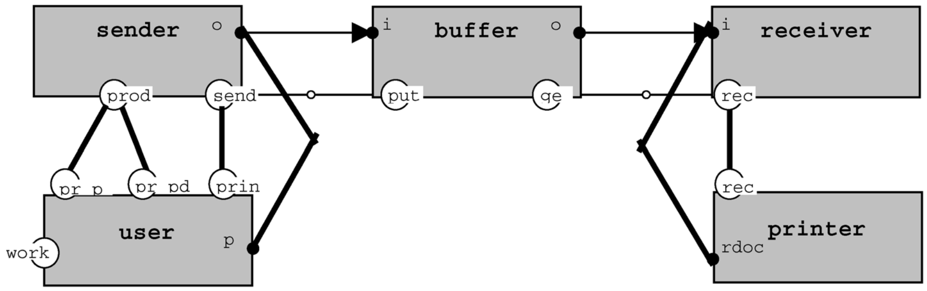
Moreover, the categorical formalisation makes it possible to prove that the design that results from the semantics of the instantiation is a refinement of the semantics of the connector itself.

As an example, let us consider, for simplicity, a connector with one role.

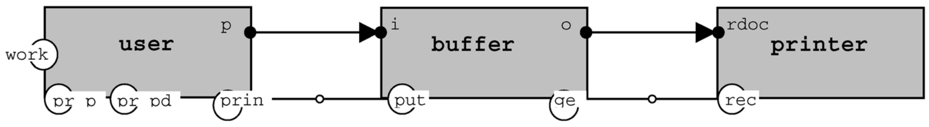


The meaning of the connector is given by the colimit of the pair $<\mu,\sigma>$ – $<\alpha{:}R\rightarrow C,\beta{:}G\rightarrow C>$. The instantiation of the role with the component P through the refinement morphism ϕ is given by the colimit of $<\mu;\phi,\sigma>$ – $<\alpha'{:}P\rightarrow S,\beta'{:}G\rightarrow S>$. We can prove that there exists a refinement morphism $\phi'{:}C\rightarrow S$, which establishes the "correctness" of the instantiation mechanism. This is because all the different objects and morphisms involved can be brought into a more general category in which the universal properties of colimits guarantee the existence of the required refinement morphism. A full proof of this property can be found in [21].

As an example, consider again the connector *Async*. As we have already seen, its roles – *sender* and *receiver* – are refined by the designs *user* and *printer*, respectively. Therefore, *Async* can be used for interconnecting these two components, giving rise to a configuration in which *user* sends the print requests to *printer* through a bounded channel.



The final configuration is obtained by calculating the composition of the signature morphisms that define the two connections of *Async* with the refinement morphisms η,κ. For instance, the channel *p* of *user* gets connected to the input channel *i* of *buffer* because $\eta(o)=p$ and *o* is connected to *i* of *buffer*. The resulting configuration is exactly the one we have already presented in section 2.2.



Architectural connectors are used for systematising software development by offering "standard" means for interconnecting components that can be reused from one application to another. In this sense, the "typical" glue is a program that implements a well-established pattern of behaviour that can be superposed to existing components of a system through the instantiation of the roles of the connector.

However, architectures also fulfil an important role in supporting a high-level description of the organisation of a system by identifying its main components and the way these components are interconnected. An early identification of the architectural elements intended for a system will help to manage the subsequent design phases according to the organisation that they imply, identifying opportunities for reuse or the integration of third-party components. From this point of view, it seems useful to allow for connectors to be based on glues that are not yet fully developed as programs but for which concrete commitments have already been made to determine the type of interconnection that they will ensure. For instance, at an early stage of development, one may decide on adopting a client-server architecture without committing to a specific protocol of communication between the client and the server. This is why, in the definition of connector in CommUnity, we left open the possibility for the glue not to be a program but a design in general.

However, in this more general framework, we have to account for the possible refinements of the glue. What happens if we refine the glue of a connector that has been instantiated to given components of a system? Is the resulting design a refinement of the more abstract design from which we started? More generally, how do connectors propagate through design, be it because the instances of the roles are refined or the glue is refined? One of the advantages of using Category Theory as a mathematical framework for formalising architectures is that answers to questions like these can be discussed at the right level of abstraction. Another advantage is that the questions themselves can be formulated in terms that are independent of any specific ADL and answered by characterising the classes of ADLs that satisfy the given properties. This is what we will do in later sections.

3.2 Examples

We now present more examples of connectors, namely some that we will need in later sections for illustrating algebraic operations on connectors. These examples relate to a case study on mobility [29]: One or more carts move continuously in the same direction on a *U*-units long circular track. A cart advances one unit at each step. Along the track there are stations. There is at most one station per unit. Each station corresponds to a check-in counter or to a gate. Carts take bags from check-in stations to gate stations. All bags from a given check-in go to the same gate. A cart transports at most one bag at a time. When it is empty, the cart picks a bag up from the nearest check-in. Carts must not bump into each other. Carts also keep a count of how many laps they have done, starting at some initial location.

The program that controls a cart is

```
design cart is
in   idest: 0..U - 1, ibag : int
out  obag, laps : int
prv  loc: 0..U - 1, dest: -1..U - 1, initloc : int
do   move: loc ≠ dest → loc := loc +_U 1 ‖  laps := if(loc=initloc,laps+1,laps)
[]   get: dest = -1 → obag := ibag ‖ dest := idest
[]   put: loc = dest → obag := 0 ‖ dest := -1
```

where $+_U$ is addition modulo U.

Locations are represented by integers from zero to the track length minus one. Bags are represented by integers, the absence of a bag being denoted by zero. Whenever the cart is empty, its destination is an impossible location, so that the cart keeps moving until it gets a bag and a valid gate location through action *get*. When it reaches its destination, the cart unloads the bag through action *put*. Notice that since input channels may be changed arbitrarily by the environment, the cart must copy their values to output channels to make sure the correct bag is unloaded at the correct gate.

A check-in counter manages a queue of bags that it loads one by one onto passing carts.

```
design check-in is
out  bag : int, dest : 0..U - 1,
prv  loc : 0..U - 1, next : bool, q : list(int)
do   new: q≠[] ∧ next → bag := head(q) || q := tail(q) || next := false
[]   put: ¬next → next := true
```

Channel *next* is used to impose sequentiality among the actions. In a configuration in which a cart is loading at a gate, the *put* action must be synchronised with a cart's *get* action and channels *bag* and *dest* must be shared with *ibag* and *idest*, respectively.

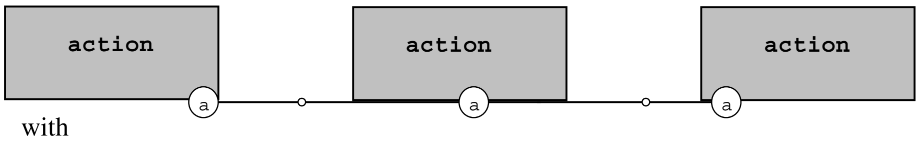
A gate keeps a queue of bags and adds each new bag to the tail.

```
design gate is
in   bag : int
prv  loc : 0..U - 1, q : list(int)
do   get : q := q.bag
```

In a configuration in which a cart is unloading at a gate, action *get* of the gate must be synchronised with the cart's *put* action, and channel *bag* must be shared with *obag*.

Synchronisation

We begin with the connector that allows us to synchronise two actions of different components. A cable would suffice for this purpose, but it is not able to capture the general case of transient synchronisation [29]. Having already a connector for the simpler case makes the presentation more uniform. The synchronisation connector has a glue identical to its roles:



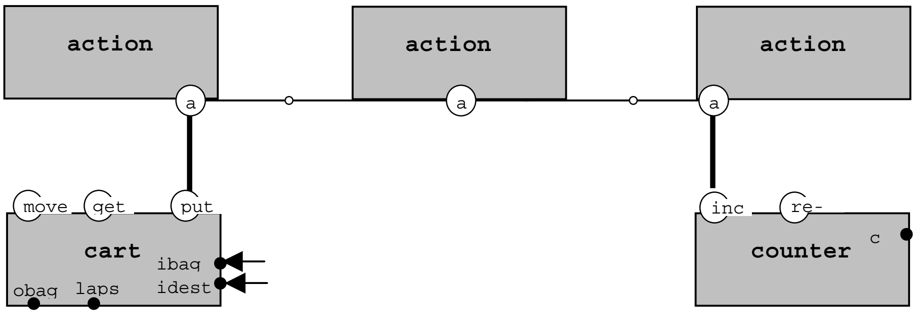
with

```
design action is
do   a: true,false → skip
```

Notice that the action has the least deterministic specification possible: its guard is given the widest possible interval and no commitments on its effects. Hence, it can be refined by any action.

According to the colimit semantics of connectors, when the two roles are instantiated with particular actions a_1 and a_2 of particular components, the components have to synchronise with each other every time one of them wants to execute the corresponding action: either both execute the joint action, or none executes.

As an example of using this connector, if we wish to count how often a cart unloads, we can monitor its *put* action with a counter:



```
design counter is
out  c:int
do   inc: true → c := c + 1
▯    reset: true → c := 0
```

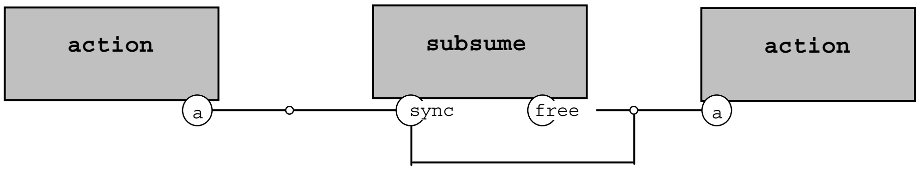
Notice that, because *inc* is always enabled, its synchronisation with *put* does not interfere with the behaviour of the cart. Hence, we can say that the counter is, indeed, *monitoring* the cart.

Subsumption

Intuitively, synchronisation is an "equivalence" between the occurrence of two actions: the occurrence of each of the actions "implies" the occurrence of the other. In certain circumstances, we are interested in connecting two actions by only one of the implications. For instance, to avoid a cart c_1 colliding with the cart c_2 right in front of it, we only need one implication: if c_1 moves, so must c_2. The other implication is not necessary. The analogy with implication also extends to the counter-positive: if c_2 cannot move, for instance because it is (un)loading a bag, then neither can c_1. We call this "one-way" synchronisation *action subsumption.*

As for the connector, it simply adds to the synchronisation connector the ability to let the subsumed action occur freely. This is only possible because our signature morphisms are contravariant on actions and, hence, allow an action to ramify into a set of actions. In order to prevent carts from colliding, all that we have to do is to ramify the move action of the cart in front in two: one that accounts for the situation in which the cart behind moves, which needs to be performed synchronously, and the other to account for free move actions, i.e. moves that are not implied by the cart behind.

The subsumption connector is as follows:

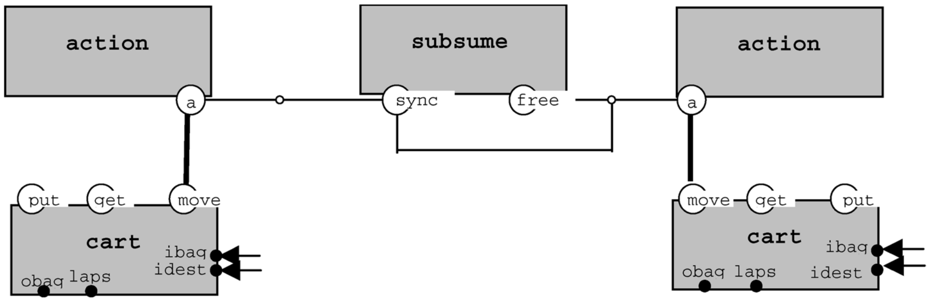


where the glue is now given by

```
design subsume is
do  sync: true,false → skip
[]  free: true,false → skip
```

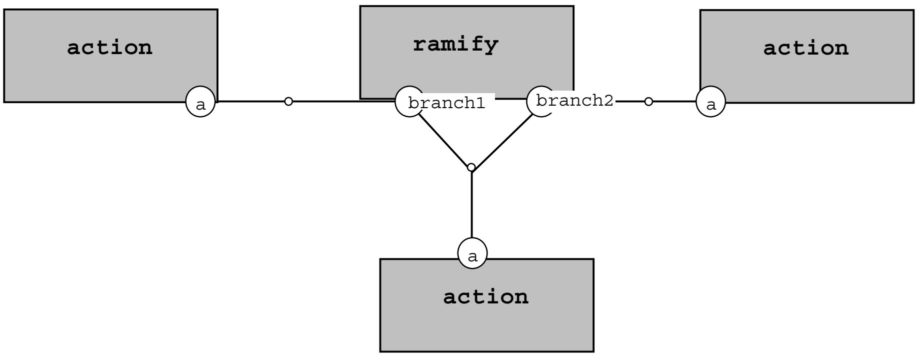
Notice that although the two roles are equal, the binary connector is not symmetric because the connections treat the two actions differently: the right-hand one may be executed alone at any time, while the left-hand one must co-occur with the right-hand one, through action *sync*. Indeed, if we instantiate the left-hand role with an action a_1 and the right-hand role with an action a_2, the semantics of the interconnection, as obtained through the colimit, is given by two synchronisation sets: $\{a_1,sync,a_2\}$ and $\{free,a_2\}$. Notice that action a_1 can only occur together with a_2 but that a_2 can occur without a_1.

Hence, the left-hand action is the one that we want to connect to the *move* action of the cart behind – c_1, while the right-hand action is associated to the movement of c_2 – the cart in front, as shown next.



Ramification

A generalisation of the previous connectors is to allow an action to synchronise, independently, with two actions of two different programs. The importance of this connector will become apparent in Section 5, where it is shown to be a primitive from which other connectors can be built.

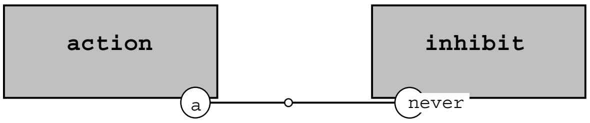


```
design ramify is
do   branch1: true, false → skip
     branch2: true, false → skip
```

If we instantiate the left-hand role with an action a_1 and the right-hand role with an action a_2, and the middle role with an action b, the semantics of the interconnection, as obtained through the colimit, is given by two synchronisation sets: *{a_1,branch$_1$,b}* and *{a_2,branch$_2$,b}*. Notice that action b will always occur simultaneously with either a_1 or a_2 but not with both. This connector can be generalised to any finite number of ramifications. It allows for a server, e.g. a buffer, to be connected simultaneously, but independently, to a fixed maximum number of components.

Inhibition

Another useful connector is the one that allows us to inhibit an action by making its guard false. This is useful when, for some reason, we need to prevent an action from occurring but without having to reprogram the component. Indeed, the mechanism of superposition that we have used as a semantics for the application of architectural connectors allow us to disable an action without changing the guard directly: it suffices to synchronise the action with one that has a false guard.



```
design inhibit is
do never: false → skip
```

When the role is instantiated with an action with guard B, the result of the interconnection is the same action guarded by $B \wedge$***false***. This connector can be generalised to arbitrary conditions with which one wants to strengthen the guards of given actions. The inhibitor just has to be provided with the data that is necessary to compute the condition C that will strengthen the guard, for instance through the use of input channels through which we can select the sources of the information that will disable the action.

```
design inhibit(C) is
in ...
do never: C → skip
```

The result of instantiating the role with an action with guard B is the same action guarded by $B \wedge C$.

4 An ADL-Independent Notion of Connector

The notion of connector that we presented for CommUnity can be generalised to other design formalisms. In this section, we shall discuss the properties that such formalisms need to satisfy for supporting the architectural concepts and mechanisms that we have illustrated for CommUnity.

Before embarking on this discussion, we need to fix a framework in which designs, configurations and relationships between designs, such as refinement, can be formally described. Our past experience in formalising notions of structure in Computing, building on previous work of J.Goguen on General Systems Theory, suggests that, as illustrated in section 2, Category Theory provides a convenient framework for our purpose. More concretely, we shall consider that a formalism supporting system design includes :

- a category ***c-DESC*** of component descriptions in which systems of interconnected components are modelled through diagrams;
- for every set *CD* of component descriptions, a set *Conf(CD)* consisting of all well-formed configurations that can be built from the components in *CD*. Each such configuration is a diagram in ***c-DESC*** that is guaranteed to have a colimit. Typically, *Conf* is given through a set of rules that govern the interconnection of components in the formalism.
- a category ***r-DESC*** with the same objects as ***c-DESC***, but in which morphisms model refinement, i.e. a morphism $\eta: S \rightarrow S'$ in ***r-DESC*** expresses that S' refines S, identifying the design decisions that lead from S to S'. Because the description of a composite system is given by a colimit of a diagram in ***c-DESC*** and, hence, is defined up to an isomorphism in ***c-DESC***, refinement morphisms must be such that descriptions that are isomorphic in ***c-DESC*** refine, and are refined exactly by, the same descriptions. Hence, it is required that $\mathit{Isomorph}(\textbf{\textit{c-DESC}}) \unlhd \mathit{Isomorph}(\textbf{\textit{r-DESC}})$.

Summarising, all that we require is a notion of system description, a relationship between descriptions that captures components of systems, another relationship that captures refinement, and criteria for determining when a diagram of interconnected components is a well-formed configuration. These requirements are discussed in more detail in [23].

4.1 Architectural Schools

In the context of this categorical framework, we shall now discuss the properties that are necessary for supporting Software Architectures.

Coordination

A key property of a formalism for supporting architectural design is that it provides a clear separation between the description of individual components and their interaction in the overall system organisation. In other words, the formalism must support the separation between what, in the description of a system, is responsible for its computational aspects and what is concerned with coordinating the interaction between its different components.

In the case of CommUnity, as we have seen, only signatures are involved in interconnections. The body of a component design describes its functionality and, hence, corresponds to the computational part of the design. At the more general level that we are discussing, we shall take the separation between coordination and computation to be materialised through a functor ***sig: c-DESC→SIG*** mapping descriptions to signatures, forgetting their computational aspects. The fact that the computational side does not play any role in the interconnection of systems can be captured by the following properties of this functor:

- ***sig*** *is faithful;*
- ***sig*** *lifts colimits of well-formed configurations;*
- ***sig*** *has discrete structures;*

together with the following condition on the well-formed configuration criterion

- *given any pair of configuration diagrams* $\boldsymbol{dia_1}$, $\boldsymbol{dia_2}$ *s.t.* $\boldsymbol{dia_1;sig=dia_2;sig}$, *either both are well-formed or both are ill-formed.*

The fact that ***sig*** is faithful (i.e., injective over each hom-set) means that morphisms of systems cannot induce more relationships than those that can be established between their underlying signatures.

The requirement that sig lifts colimits means that, given any well-formed configuration expressed as a diagram ***dia:I→c-DESC*** of descriptions and colimit $(\boldsymbol{sig}(S_i)\rightarrow\theta)_{i:I}$ of the underlying diagram of signatures, i.e. of *(**dia;sig**)*, there exists a colimit $(S_i\rightarrow S)_{i:I}$ of the diagram ***dia*** of descriptions whose signature part is the given colimit of signatures, i.e. $\boldsymbol{sig}(S_i\rightarrow S)=(\boldsymbol{sig}(S_i)\rightarrow\theta)$. This means that if we interconnect system components through a well-formed configuration, then any colimit of the underlying diagram of signatures establishes a signature for which a computational part exists that captures the joint behaviour of the interconnected components.

The requirement that ***sig*** has discrete structures means that, for every signature θ:***SIG*,** there exists a description $s(\theta)$:***c-DESC*** such that, for every signature morphism $f:\theta\rightarrow\boldsymbol{sig}(S)$, there is a morphism $g:s(\theta)\rightarrow S$ in ***c-DESC*** such that $\boldsymbol{sig}(g)=f$. That is to say, every signature θ has a "realisation" (a discrete lift) as a system component $s(\theta)$ in the sense that, using θ to interconnect a component S, which is achieved through a morphism $f:\theta\rightarrow\boldsymbol{sig}(S)$, is tantamount to using $s(\theta)$ through any $g:s(\theta)\rightarrow S$ such that $\boldsymbol{sig}(g)=f$. Notice that, because ***sig*** is faithful, there is only one such g, which means that f and g are, essentially, the same. That is, sources of morphisms in diagrams of descriptions are, essentially, signatures.

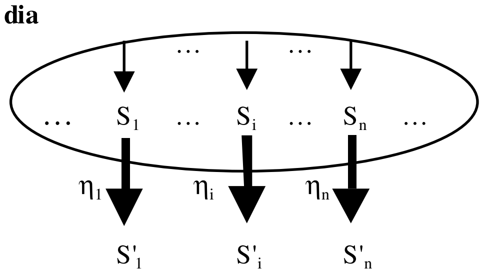
These properties constitute what we call a *coordinated formalism*. More precisely, we say that ***c-DESC*** *is coordinated over* ***SIG*** *through the functor* ***sig***. In a coordinated formalism, any interconnection of systems can be established via their signatures, legitimating the use of signatures as cables in configuration diagrams. By requiring that any two configuration diagrams that establish the same interconnections at the level of signatures be either both well-formed or both ill-formed, the fourth property ensures that the criteria for well-formed configurations do not rely on the computational parts of descriptions.

Refinement and Compositionality

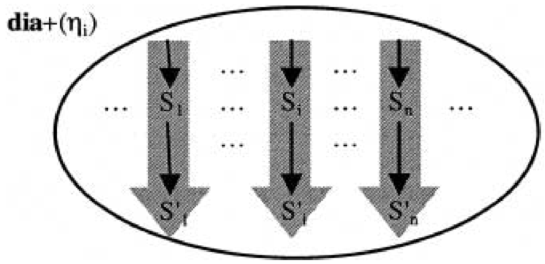
Another crucial property for supporting architectural design is in the interplay between structuring systems in architectural terms and refinement. We have already pointed out that one of the goals of Software Architectures is to support a view of the gross organisation of systems in terms of components and their interconnections that can be carried through the refinement steps that eventually lead to the implementation of all its components. Hence, it is necessary that the application of architectural connectors to abstract designs, as a means of making early decisions on the way certain components need to be coordinated, will not be jeopardised by subsequent refinements of the component designs towards their final implementations. Likewise, it is desirable that the application of a connector may be made on the basis of an abstract design of its glue as a means of determining main aspects of the required coordination without committing to the final mechanisms that will bring about that coordination.

One of the advantages of the categorical framework that we have been proposing is that it makes the formulation of these properties relatively easy, leading to a characterisation of the design formalisms that support them in terms of the structural properties that we have been discussing. For instance, we have already seen that, in the situations in which refinement morphisms map directly to signature morphisms, we may simply put together, in a diagram of signatures, the morphisms that define the interactions and the morphisms that establish the refinement of the component descriptions.

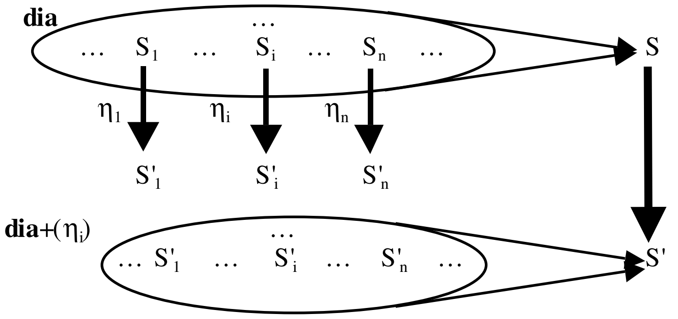
More precisely, in the situations in which there exists a forgetful functor ***r-sig*: *r-DESC*$\rightarrow$*SIG*** that agrees with the coordination functor ***sig*** on signatures, i.e. ***r-sig(S)=sig(S)*** for every *S:**c-DESC***, given a well-formed configuration diagram ***dia*** of a system with components $S_1,\ldots,S_n$ and refinement morphisms $\eta_i{:}S_i\rightarrow S'_i$: $i\in 1..j$,



we can obtain a new diagram in ***c-DESC*** and, hence, a new configuration, by composing the morphisms ***r-sig(***η_i***)*** with those in ***dia*** that originate in cables (signatures) and have the S_i as targets.



However, as we have seen in the case of CommUnity, it may be possible to propagate the interactions between the components of a system when their descriptions are replaced by more concrete ones even when refinement morphisms do not map to signature morphisms. This more general situation can be characterised by the existence, for every well-formed configuration ***dia*** involving descriptions $\{S_1,...,S_n\}$ and refinements morphisms $\{\eta_i:S_i\rightarrow S'_i: i\in 1..n\}$, of a well-formed configuration diagram ***dia***+(η_i) that characterises the system obtained by replacing the S_i by their refinements. The correctness criterion for this form of "configuration refinement" is that the colimit of ***dia***+(η_i) provides a refinement for the colimit of ***dia***.



We shall say that the formalisms that support such forms of correct configuration refinement are *compositional.*

When we consider the specific case of the configurations obtained by direct instantiation of an architectural connector, this property reflects the compositionality of the connector as an operation on configurations. Compositionality ensures that the semantics of the connector is preserved (refined) by any system that results from its instantiation. For instance, in the case of a binary connector

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \textbf{dia} & \theta_2 \arrow[ul] \arrow[d] \\
R_1 & & R_2
\end{tikzcd}$$

and given instantiations $\eta_1{:}R_1 \rightarrow P_1$ and $\eta_2{:}R_2 \rightarrow P_2$, the description returned by the colimit of ***dia*** is refined by the description returned by the colimit of $\boldsymbol{dia}+(\eta_1,\eta_2)$.

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \textbf{dia}+(\eta_1,\eta_2) & \theta_2 \arrow[ul] \arrow[d] \\
P_1 & & P_2
\end{tikzcd}$$

Likewise, compositionality guarantees that if a connector with an abstract glue *G* is applied to given designs, and the glue is later on refined through a morphism $\eta{:}G \rightarrow G'$, the description that is obtained through the colimit of $\boldsymbol{dia}+\eta$ is a refinement of the semantics of the original instantiation. In fact, we can consider the refinement of the glue to be a special case of an operation on the connector that delivers another connector – a refinement of the original one in the case at hand. Other operations will be analysed in section 5.

Summary

*In summary, a formalism F=<**c-DESC,Conf,r-DESC**> supports architectural design, and is called an architectural school, if*

- ***c-DESC** is coordinated over a category **SIG** through a functor **sig: c-DESC→SIG***
- *F is compositional*

GAMMA

We shall now illustrate the notion of architectural school with an example borrowed from coordination formalisms: the language Gamma [5] based on the chemical reaction paradigm.

A Gamma program P consists of

- *a signature Σ=<S,Ω,F>, where S is a set of sorts, Ω is a set of operation symbols and F is a set of function symbols, representing the data types that the program uses;*
- *a set of reactions, where a reaction R has the following structure:*

$$R \equiv X, t_1, \ldots, t_n \rightarrow t'_1, \ldots, t'_m \Leftarrow c$$

where

- *X is a set (of variables); each variable is typed by a data sort in S;*
- *$t_1, ..., t_n \rightarrow t'_1, ..., t'_m$ is the action of the reaction – a pair of sets of terms over X;*
- *c is the reaction condition – a proposition over X.*

An example of a Gamma program is the following producer of burgers and salads from, respectively, meat and vegetables:

```
PROD ≡  sorts     meat, veg, burger, salad
        ops       vprod: veg→salad, mprod: meat→burger
        reactions m:meat, m → mprod(m)
                  v:veg, v →•vprod(v)
```

Gamma programs are always composable and their parallel composition, as defined in [5], is a program consisting of all the reactions of the component programs – its behaviour is obtained by executing the reactions of the component programs in any order, possibly in parallel. This leads us to the following notion of morphism.

A morphism σ between Gamma programs P_1 and P_2 is a morphism between the underlying data signatures s.t. $\sigma(P_1) \subseteq P_2$, i.e., P_2 has more reactions than P_1. Gamma programs and their morphisms constitute a category **GAMMA**.

This category is well behaved as far as interconnections are concerned, which means that every diagram is considered to be a well-formed configuration. In order to illustrate system configuration in Gamma, consider that we want to interconnect the producer with the following consumer:

```
CONS  ≡ sorts     food, waste
        ops       cons: food→waste,
        reactions f:food, f→cons(f)
```

The interconnection of the two programs is based on the identification of the food the consumer consumes, that is, the interconnection is established between their data types. For instance, the coordination of the producer and the consumer based on meat is given by the following interconnection:

$$\begin{tikzcd} & \textit{sorts}\ \texttt{s} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\texttt{meat}\leftarrow\texttt{s}\rightarrow\texttt{food}"] & \\ \texttt{PROD} & \ & \texttt{CONS} \end{tikzcd}$$

Given the simplicity of the mechanisms available in Gamma for interconnecting components, it is not difficult to conclude that Gamma is coordinated over the category of data types:

- the forgetful functor *dt* from Gamma programs to data types is faithful;
- given any diagram in the category ***GAMMA***, a colimit $\sigma_i:(dt(P_i)\rightarrow\Sigma)_{i:I}$ of the corresponding diagram in the category of data types is lifted to the following colimit of programs $\sigma_i:(P_i\rightarrow<\Sigma,\cup\sigma_j(R_j)>)_{i:I}$;
- the discrete lift of a data type is the program with the empty set of reactions.

Notice, however, that we have extended the way in which Gamma programs are traditionally put together. Gamma assumes a global data space whereas we have made it possible for Gamma programs to be developed separately and put together by matching the features that they are required to have in common. This localisation further enhances the reusability of Gamma programs.

In order to complete the picture, it remains to provide a notion of refinement. This is very simple because the morphisms of ***GAMMA*** may also be used for modelling the refinement relationship. That is to say, a program *P* is refined by a program *Q* if and only the reactions of *P* are also present in Q. It is very easy to check that ***GAMMA*** defined in this way is compositional and, hence, defines an architectural school.

Besides CommUnity and Gamma, other formalisms define architectural schools. For instance, the concurrency models that are formalised in [28] using categorical techniques satisfy the properties that we have laid down for architectural schools. Such formalisms fulfil a very important role in providing models of system behaviour that can be used as abstractions of non-software components in architectures. The category of theories of any institution [19] also defines an architectural school, thus showing that another class of abstractions – capturing specifications of system behaviour – can also be used for architectural design. Hence, the notion of architectural school that we put forward encompasses a large variety of formalisms. In the next sections, we are going to show how we can capitalise on this variety.

4.2 Adding Abstraction to Architectural Connectors

The mathematical framework that we presented in the previous sections provides not only an ADL-independent semantics for the principles and techniques that can be found in existing approaches to Software Architectures, but also a basis for extending the capabilities of existing ADLs. Until the end of the paper, we will present and explore some of the avenues that this mathematical characterisation has opened, hoping that the reader will want to explore them even further. In this section, we will show how the roles of a connector can be formulated at a more declarative level, such as a specification logic, instead of a design language like CommUnity.

As already mentioned, the purpose of the roles in a connector is to impose restrictions on the local behaviour of the components that are admissible as instances. In the approach to architectural design outlined in the previous sections, this is achieved through the notion of correct instantiation via refinement morphisms. As also seen above, roles do not play any part in the calculation of the resulting system. They are only used for defining what a correct instantiation is. This separation of concerns justifies the adoption of a more declarative formalism for the specification of roles, namely one in which it is easier to formulate the properties required of components to be admissible instances.

In this section, we are going to place ourselves in the situation in which the glues are designs, the roles are specifications, and the instantiations of the roles are, again, designs. We are going to consider that specifications are given as a category ***SPEC***, e.g. the category of theories of a logic formalised as an institution [19]. We take the relationship between specifications and designs to be captured through the following elements:

- a functor ***spec:SIG→SPEC*** mapping signatures and their morphisms to specifications.
- a satisfaction relation $\models$ between design morphisms and specification morphisms satisfying the following properties:

- *If $\pi:P\to P' \models \sigma:S\to S'$, then $id_P \models id_S$* and *$id_{P'} \models id_{S'}$.*
- *If $\pi_1:P_1\to P_2 \models \sigma_1:S_1\to S_2$ and $\pi_2:P_2\to P_3 \models \sigma_2:S_2\to S_3$ then $\pi_1;\pi_2 \models \sigma_1;\sigma_2$*
- *Let **s**:**I**→**SPEC** be a diagram of specifications and **p**:**I**→**c-DSGN** a diagram of designs with the same shape such that, for every edge $f:i\to j$ in **I**, $\mathbf{p}_f:\mathbf{p}_i\to\mathbf{p}_j \models \mathbf{s}_f:\mathbf{s}_i\to\mathbf{s}_j$. We require that, if **p** admits a colimit $\pi_i:\mathbf{p}_i\to P$, then **s** admits a colimit $\sigma_i:\mathbf{s}_i\to S$ such that, for every node i:**I**, $\pi_i \models \sigma_i$.*
- *If $id_P \models id_S$ and $\rho:P\to P'$ is a refinement morphism, then $id_{P'} \models id_S$*
- *For every signature θ, $id_{dsgn(\theta)} \models id_{spec(\theta)}$*

The idea behind the functor ***spec*** is that, just like, through ***dsgn***, signatures provide the means for interconnecting designs, they should also provide means for interconnecting specifications. Hence, every signature generates a canonical specification – the specification of a cable. However, we do not put as many constraints on ***spec*** as on ***dsgn*** because, for the purposes of this section, we are limiting the use of specifications to the definition of connector roles. Naturally, if we wish to address architecture building at the specification level, then we will have to require ***SPEC*** to satisfy the properties that we discussed in section 4.1.

The satisfaction relation is defined directly on morphisms because our ultimate goal is to address interconnections, not just components. Satisfaction of component specifications by designs is given through the identity morphisms. The properties required of the satisfaction relation address its compatibility with the categorical constructions that we use, namely composition of morphisms and colimits. The last two properties mean that refinement of component designs leaves the satisfaction relation invariant, and that the design (cable) generated by every signature satisfies the specification (cable) generated by the same signature.

Given such a setting, we generalise the notion of connector as follows:

- *A connection consists of*
  - *a design G and a specification R, called the glue and the role of the connection, respectively;*
  - *a signature θ and two morphisms μ:**dsgn**(θ)→G, σ:**spec**(θ)→R in **c-DSGN** and **SPEC**, respectively, connecting the glue and the role via the signature (cable).*
- *A connector is a finite set of connections with the same glue.*
- *An instantiation of a connection with signature θ and role morphism σ consists of a design P and a design morphism π:**dsgn**(θ)→P such that $\pi \models \sigma$.*
- *An instantiation of a connector consists of an instantiation for each of its connections. An instantiation is said to be correct if the diagram defined by the instantiation morphisms and the glue morphisms is a well-formed configuration. The colimit of this configuration defines the semantics of the instantiation, guaranteed to exist if the instantiation is correct.*

Although the generalisation seems to be quite straightforward, we do not have an immediate generalisation for the semantics of connectors. This is because the glue is a design and the role is a specification, which means that a connector does not provide us with a diagram like in the homogeneous case that we studied in section 3. However, if we are provided with a specification for the glue, we can provide semantics for the connector at the specification level:

- *A complete connection consists of*
  - *a design G and a specification R, called the glue and the role of the connection, respectively;*
  - *a signature θ and two morphisms μ:**dsgn**(θ)→G, σ:**spec**(θ)→R in **c-DSGN** and **SPEC**, respectively, connecting the glue and the role via the signature (cable);*
  - *a specification S and a morphism τ:**spec**(θ)→S such that $\mu \models \tau$. Note that this means that the design G satisfies the specification S.*
- *A complete connector is a finite set of complete connections with the same glue design and specification. Its semantics is given by the colimit, if it exists, of the **SPEC**-diagram defined by the σ_i and the τ_i.*
- *The semantics of the instantiation of a complete connector satisfies the semantics of the connector.*

An illustration of an abstract architectural school can be given in terms of a linear temporal logic. We adopt the traditional notion of specification as a collection of sentences (a theory presentation) expressing the properties the component is required to satisfy. These properties are given in terms of the vocabulary of the component being specified – a set of input channels, a set of output channels and a set of action names.

We start by defining the language of the logic, for which the distinction between input and output channels is irrelevant, and then we define specifications.

A temporal signature is a pair $<V,\Gamma>$ where

- *V is an S-indexed family of mutually disjoint finite sets,*
- *Γ is a finite set.*

The language of terms over a temporal signature $t\theta=<V,\Gamma>$, with sort s, is $Term(t\theta)_s$ defined by

$$t_s ::= v \mid c \mid (\mathbf{X} t_s) \mid f(t_{s_1},\dots,t_{s_n})$$

for every $v\in V_s$, $c\in\Omega_{<>,s}$ and $f\in\Omega_{<s_1,\dots,s_n>,s}$. The language of temporal propositions over a temporal signature $t\theta=<V,\Gamma>$ is $Prop(t\theta)$ defined by

$$\phi ::= (t_1 =_s t_2) \mid g \mid (\neg\phi) \mid (\phi\supset\phi') \mid (\mathbf{X}\phi) \mid (\phi\mathbf{U}\phi')$$

for every $s\in S$, $g\in\Gamma$, $t_1,t_2\in Term(t\theta)_s$.

The temporal operators are ***X*** (next) and ***U*** (until). Intuitively,

- ***X****ϕ* holds in a state if ϕ holds in the next state;
- *ϕ**U**ϕ'* holds when ϕ' will hold sometime in the future and ϕ holds between now and then.

As usual, we will consider that other temporal operators like ***G*** (always in the future) and ***F*** (eventually) are defined as abbreviations: $\mathbf{G}\phi \equiv_{abv} (\phi\mathbf{U}false)$ and $\mathbf{F}\phi \equiv_{abv} (\neg\mathbf{G}(\neg\phi))$.

A specification is a pair $<<V,\Gamma,tv>,\Phi>$ where $<V,\Gamma>$ is a temporal signature, $tv:V\to\{in,out\}$ is a total function (that identifies the type of channels*) and Φ is a set of propositions in $Prop(<V,\Gamma>)$.*

As an example, we present below the specifications of a typical sender and receiver of messages through a pipe.

```
design sender[t:sort] is                      design receiver[t:sort] is
out      o:t, eof:bool               in       o:t, eof:bool
actions       send                                      out   cl:bool
axioms        eof ⊃ G(¬send ∧ eof)                      actions  rec
                                     axioms   cl ⊃ G(¬rec ∧ cl)
                                              ((eof⊃Geof) ∧ (eof∧ cl)) ⊃ (¬recUcl)
```

The component *sender* interacts with its environment through the action *send* accounting for the transmission of a message through the output channel *o*. Furthermore, it signals the end of data through the output channel *eof*. The axiom requires that *eof* be stable (remains true once it becomes true) and transmission of messages to cease once *eof* becomes true.

The component *receiver* interacts with its environment through the action *rec* accouting for the reception of a message through the input channel *i*. The other means of interaction with the environment is concerned with the closure of communication. The component receives, through the input channel *eof*, a Boolean that indicates if transmission along *i* has ceased and signals the closure of communication through the output channel *cl*. The first axiom requires that *cl* be stable and the reception of messages to cease once *cl* becomes true. The second axiom expresses that, if the information received through *eof* is stable, the receiver is obliged to close the communication as soon as it is informed that there will be no more data. However, the receiver may decide to close the communication before that.

Specification morphisms are taken as property preserving mappings that, as happens with the morphisms of design signatures, must map output channels into output channels.

A morphism $\sigma:t\theta_1\to t\theta_2$ of temporal signatures is a pair $<\sigma_{ch},\sigma_{ac}>$ where $\sigma_{ch}:V_1\to V_2$ is a total function satisfying $sort_2(\sigma_{ch}(v))=sort_1(v)$, for every $v\in V_1$, and *$\sigma_{ac}: \Gamma_2\to\Gamma_1$ is a partial mapping.*

A morphism $\sigma:<<V_1,\Gamma_1,tv_1>,\Phi_1>\to<<V_2,\Gamma_2,tv_2>,\Phi_2>$ of specifications is a morphism $\sigma:<V_1,\Gamma_1>\to<V_2,\Gamma_2>$ of temporal signatures s.t.

1. *if $tv_1(v)=out$ then $tv_2(\sigma_{ch}(v))=out$*
2. *$\Phi_2 \vdash \underline{\sigma}(\Phi_1)$*

where $\vdash$ is the usual consequence relation for linear temporal logic and $\underline{\sigma}$ is the extension of σ to the language of propositions. Specifications and their morphisms constitute a category ***SPEC***.

It remains to capture the relationship between specifications as just introduced and designs in CommUnity.

First we notice that every design signature θ can be mapped into a temporal signature by forgetting the different classes of channels and actions, as well as the write frames of actions. Then, we notice that part of the semantics of CommUnity designs can be encoded in LTL. More concretely, we may express in LTL the following facts already mentioned in section 2.1:

- the negation of *L(g)* is a blocking condition of action *g*;
- for every local channel *v*, *D(v)* consists of the set of actions that can modify it;
- the condition *R(g)* holds in every state in which *g* is executed;

- private actions that are infinitely often enabled are guaranteed to be selected infinitely often.

A design $P=<<V,\Gamma,tv,ta,D>,<R,L,U>>$ has the following properties:

1. *$(g \supset L(g))$ for every $g\in\Gamma$*
2. *$\vee_{g\in D(v)}\ g \vee (\mathbf{X}v=v))$ for every $v\in loc(V)$*
3. *$(g \supset \tau(R(g))$ for every $g\in\Gamma$, where τ is a translation that replaces any primed channel v' by the term $(\mathbf{X}v)$*
4. *$(\mathbf{GF}U(g) \supset \mathbf{GF}g)$ for every $g\in prv(\Gamma)$*

The next step is to define a notion of refinement between specifications and designs.

A refinement of a specification $S=<V_S,\Gamma_S,tv_S>,\Phi_S>$ is a design $P=<<V_P,\Gamma_P,tv_P,ta,D>, \Delta>$ and a morphism $\eta:<V_S,\Gamma_S>\rightarrow<V_P,\Gamma_P>$ of temporal signatures s.t.

1. *$tv_S(v)=tv_P(\eta_{ch}(v))$*
2. *η_{ch} is injective*
3. *$Prop(P) \vdash \underline{\eta}(\Phi_S)$*

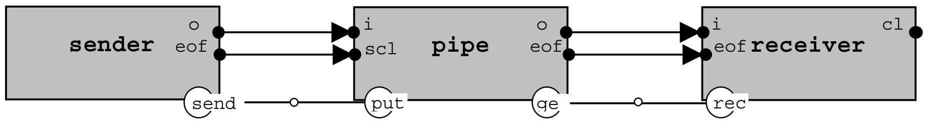
That is to say, a design refines a specification iff the design has the properties required by the specification. The signature morphism identifies for each input (resp. output) channel of the specification the corresponding input (resp. output) channel of the design and , for each action *g* of the specification, the set of actions of the design that implements *g*.

Using this notion of refinement, we can introduce a notion of satisfaction relation between design morphisms and specification morphisms:

*Given a design morphism $\pi:P\rightarrow P'$ in **c-DSGN** and a specification morphism $\sigma:S\rightarrow S'$ in **SPEC**, $\pi:P\rightarrow P' \models \sigma:S\rightarrow S'$ iff there exist refinements (P,η) of S and (P',η') of S' s.t. $\eta;\pi^*=\sigma^*;\eta'$, where π^* and σ^* are the temporal signature morphisms induced by σ and π, respectively.*

This satisfaction relation satisfies the conditions listed above.

In order to illustrate the generalised notion of connector in this setting, we present below the connector *cpipe*,



where *pipe* is the design presented below.

```
design pipe [t:sort, bound:nat] is
in      i:t, scl:bool
out     o:t, eof:bool
prv     rd: bool, b: list(t)
do      put: true → b:=b.i
▯ prv   next: |b|>0∧¬rd → o:=head(b)||b:=tail(b)||rd:=true
▯       get: rd → rd:=false
▯ prv   sig: scl∧|b|=0 → eof:=true
```

This design, which is the glue of the connector, models a buffer with unlimited capacity and a FIFO discipline. It signals the end of data to the consumer of messages as soon as the buffer gets empty and the sender of messages has already informed, through the input channel *scl*, that it will not send anymore messages.

The two roles — the specifications *sender* and *receive*r introduced before— define the behaviour required of the components to which the connector *cpipe* can be applied. It is interesting to notice that, due to the fact that LTL is more abstract than CommUnity, we were able to abstract away completely the production of messages in the role *sender*. In the design of *sender* presented in 2.1 we had to consider an action modelling the production of messages.

We can now generalise these constructions even further by letting the connections use different specification formalisms and the instantiations to be performed over components designed in different design formalisms. In this way, it will be possible to support the reuse of third-party components, namely legacy systems, as well as the integration of non-software components in systems, thus highlighting the role of architectures in promoting a structured and incremental approach to system construction and evolution.

However, to be able to make sense of the interconnections, we have to admit that all the design formalisms are coordinated over the same category of signatures. That is to say, we assume that the integration of heterogeneous components is made at the level of the coordination mechanisms, independently of the way each component brings about its computations. Hence, we will assume given

- *a family $\{\mathbf{DSGN}_d\}_{d:D}$ of categories of designs, all of which are coordinated over the same category* **SIG** *via a family of functors $\{\mathbf{dsgn}_d\}_{d:D}$,*
- *a family $\{\mathbf{SPEC}_c\}_{c:C}$ of categories of specifications together with a family $\{\mathbf{spec}_c:\mathbf{SPEC}_c \rightarrow \mathbf{SIG}\}_{c:C}$ of functors,*
- *a family $\{\models_r\}_{r:R}$ of satisfaction relations, each of which relates a design formalism d(r) and a specification formalism c(r). We do not require R to be the cartesian product D×C, i.e. there may be pairs of design and specification formalisms for which no satisfaction relation is provided.*

Given such a setting, we generalise the notion of connector as follows:

- *A connection consists of*
  - *a design formalism* **DSGN** *and a specification formalism* **SPEC**;
  - *a design G:***DSGN** *and a specification R:***SPEC**, *called the glue and the role of the connection, respectively;*
  - *a signature θ:***SIG** *and two morphisms $\mu:\mathbf{dsgn}(\theta)\rightarrow G$, $\sigma:\mathbf{spec}(\theta)\rightarrow R$ in* **DSGN** *and* **SPEC**, *respectively, connecting the glue and the role via the signature (cable).*
- *A connector is a finite set of connections with the same glue.*
- *An instantiation of a connection with specification formalism* **SPEC**, *signature θ and role morphism σ consists of*
  - *a design formalism* **DSGN'** *and a satisfaction relation $\models$ between* **DSGN'** *and* **SPEC**, *each from the corresponding family;*
  - *a design P and a design morphism $\pi:\mathbf{dsgn}(\theta)\rightarrow P$ in* **DSGN'** *such that $\pi \models \sigma$.*

- *An instantiation of a connector consists of an instantiation for each of its connections.*

Given that we now have to deal with several design formalisms, providing a semantics for the resulting configurations requires a homogeneous formalism to which all the design formalisms can be mapped. Clearly, because we want the heterogeneity of formalisms to be carried through to the implementations in order to be able to support the integration of legacy code, third-party components and even non-software components, this common formalism cannot be at the same level as designs and the mapping cannot be a translation. What seems to make more sense is to choose a behaviour model that can be used to provide a common semantics to all the design formalisms so that the integration is not performed at the level of the descriptions but of the behaviours that are generated from the descriptions. Indeed, when we talk about the integration of heterogeneous components, our goal is to coordinate their individual behaviours. An architecture should provide precisely the mechanisms through which this coordination is effected.

We have already mentioned that concurrency models can be formalised in the categorical framework that we have described. See [28] for several examples. In particular, a very simple, trace-based behaviour model that is, in some sense, minimal can be found in [13]. This model should provide a common formalism for integration.

5 Towards an Algebra of Connectors

It is not always possible to adapt components to work with the existing connectors. Even in those cases where it is feasible, a better alternative may be to modify the connectors because, usually, there are fewer connector types than components types. Moreover, most ADLs either provide a fixed set of connectors or only allow the creation of new ones from scratch, hence requiring from the designer a deep knowledge of the particular formalism and tools at hand. Conceptually, operations on connectors allow one to factor out common properties for reuse and to better understand the relationships between different connector types.

The notation and semantics of such connector operators are, of course, among the main issues to be dealt with. Our purpose in this section is to show how typical operators can be given an ADL-independent semantics by formalising them in the categorical framework that we presented in the previous sections. An example of such an operator was already given in section 3. We saw how, given a connector expressed through a configuration diagram ***dia***

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \textbf{dia} & \theta_2 \arrow[ul] \arrow[d] \\
R_1 & & R_2
\end{tikzcd}$$

and a refinement $\eta{:}G\to G'$ of its glue, we can construct through ***dia***$+\eta$ another connector that has the same roles as the original one, but whose glue is now *G'*.

$$\begin{tikzcd}
 & G' & \\
\theta_1 \arrow[ur] \arrow[d] & \textbf{dia}+\eta & \theta_2 \arrow[ul] \arrow[d] \\
R_1 & & R_2
\end{tikzcd}$$

A fundamental property of this construction, given by compositionality, is that the semantics of the original connector, as expressed by the colimit of its diagram, is preserved in the sense that it is refined by the semantics of the new connector. This means that all instantiations of the new connector are refinements of instantiations of the old one. This operation supports the definition of connectors at higher levels of abstraction by delaying decisions on concrete representations of the coordination mechanisms that they offer, thus providing for the definition of specialisation hierarchies of connector types.

In the rest of this section we present three connector transformations that operate on the roles rather than the glue. Transformations that, like above, operate at the level of the glue are more sensitive in that they interfere more directly with the semantics of the connector to which they are applied. Hence, they should be restricted to engineers who have the power, and ensuing responsabilities, to change the way connectors are implemented. Operations on the roles are less critical and can be performed more liberally by users who have no access to the implementation (glue).

5.1 Role Refinement

To tailor general-purpose connectors for a specific application, it is necessary to replace the generic roles by specialised ones that can effectively act as "formal parameters" for the application at hand. Role replacement is done in the same way as applying a connector to components: there must be a refinement morphism from the generic role to the specialised one. The old role is cancelled, and the new role morphism is the composition of the old one with the refinement morphism in the sense discussed in section 3.

Given an n-ary connector

$$\begin{tikzcd}
\textbf{dia} & & G & & \\
\theta_1 \arrow[urr] \arrow[d] & & \theta_i \arrow[u] \arrow[d] & & \theta_n \arrow[ull] \arrow[d] \\
R_1 & & R_i & & R_n
\end{tikzcd}$$

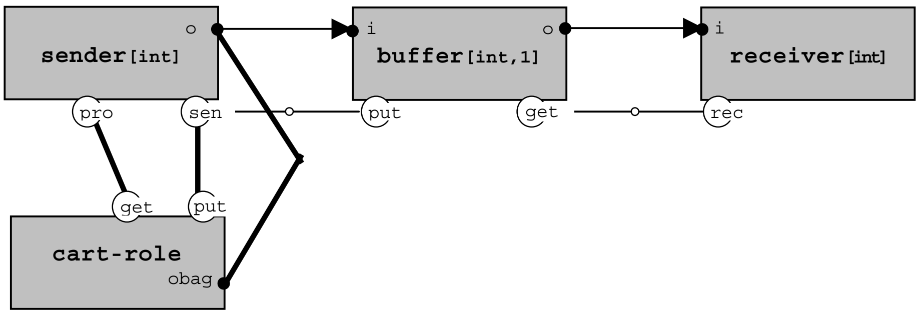
and a refinement morphism $\eta_i{:}R_i \rightarrow R'_i$ for some $0 < i \leq n$, the role refinement operation yields the connector

$$\begin{tikzcd}
\mathbf{dia}+\eta_i & G & \\
\theta_1 \arrow[ur] \arrow[d] & \theta_i \arrow[u] \arrow[d] & \theta_n \arrow[ul] \arrow[d] \\
R_1 & R'_i & R_n
\end{tikzcd}$$

Notice that this operation can be applied to both abstract and heterogeneous connectors in the sense of section 5.

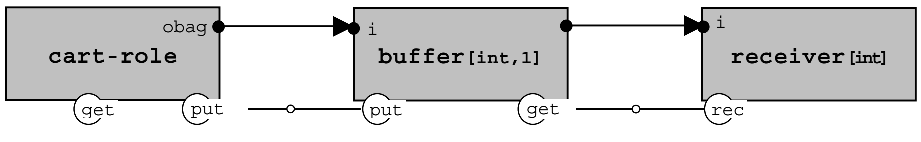
This operation preserves the semantics of the connectors to which it is applied in the sense that any instantiation of the refined connector is also an instantiation of the refined one. This is because the refinement morphism that instantiates R'_i can be composed with η_i to yield an instantiation of R_i.

As an example of role refinement, consider the asynchronous connector shown previously. This connector is too general for our luggage distribution service because the sender and receiver roles do not impose any constraints on the admissible instances. We would like to refine these roles in order to prevent meaningless applications to our example like sending the location of the check-in as a bag to the cart.



```
design cart-role is
out   obag: int
prv   dest : -1..U - 1
do    get[obag]: dest = -1 → dest' > -1
[]    put: dest>-1,false → obag := 0 || dest := -1
```

with channel *rd* of the sender refined by the term *dest ≠−1*. The resulting connector is



Notice that the invalid combinations are not possible because *cart-role* cannot be refined with a gate or a check-in, nor is it possible to refine *gate-role* with a cart or a check-in. Moreover, the *obag* channel of *cart-role* cannot be refined by channel *laps* of cart.

5.2 Role Encapsulation

To prevent a role from being further refined, the second operation we consider, when executed repeatedly, decreases the arity of a connector by encapsulating some of its roles, making the result part of the glue. This operation requires that the glue and the encapsulated role be in the same formalism.

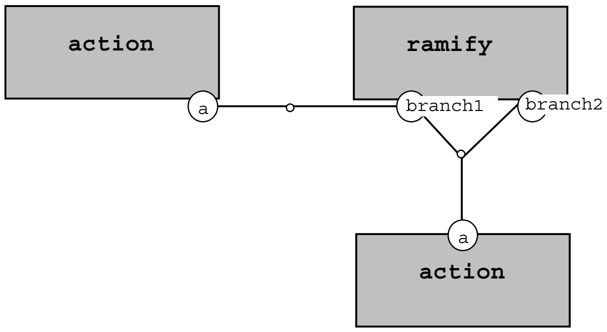
Given an n-ary connector

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \theta_i \arrow[u] \arrow[d] & \theta_n \arrow[ul] \arrow[d] \\
R_1 & R_i & R_n
\end{tikzcd}$$

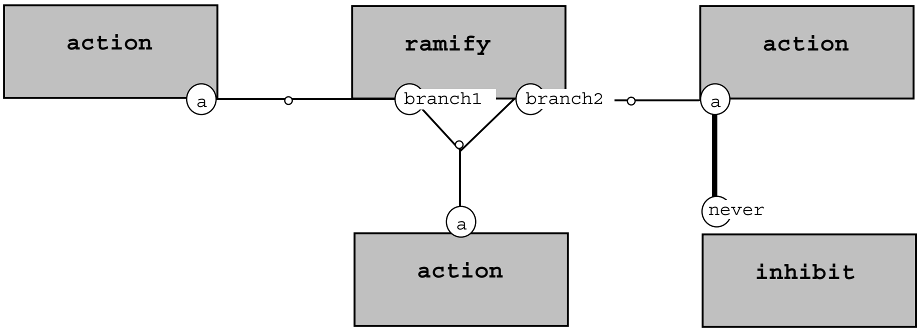
the encapsulation of the i-th role is performed as follows: the pushout of the i-th connection is calculated, and the other connections are changed by composing the morphisms that connect the cables to the glue with the morphism that connects the glue with the apex of the pushout, yielding a connector of arity n–1.

$$\begin{tikzcd}
 & G^* & \\
 & G \arrow[u] & \\
\theta_1 \arrow[ur] \arrow[uur, dashed] \arrow[d] & \theta_i \arrow[u] \arrow[d] & \theta_n \arrow[ul] \arrow[uul, dashed] \arrow[d] \\
R_1 & R_i & R_n
\end{tikzcd}$$

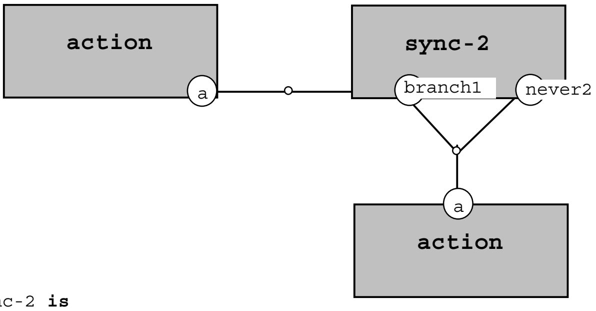
For instance, we can obtain the action subsumption connector from the action ramification one through encapsulation of the right-hand side *action* role:



We can also obtain a synchronisation connector through refinement of one role with the *inhibit* program



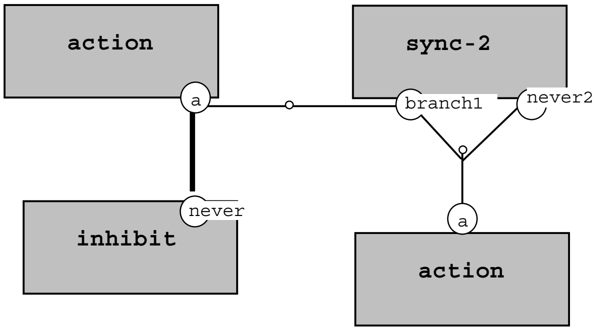
and then encapsulating it



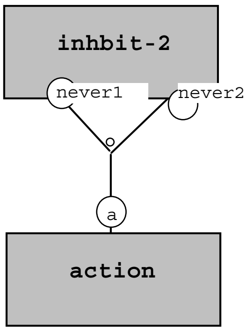
```
design sync-2 is
do   branch1: true →•skip
[]   never2: false → skip
```

Notice that the synchronisation connector obtained is not syntactically equal to the one presented in section 3.2, but it is “equivalent” because one of the ramifications of the bottom action *a* is never executed (because it is synchronized with *never2*) and thus the net effect of the connector is to synchronise both actions *a* through *branch1*.

If we now refine the left-hand branch also with *inhibit*



and then encapsulate it, we obtain a connector that is equivalent to *inhibit*.



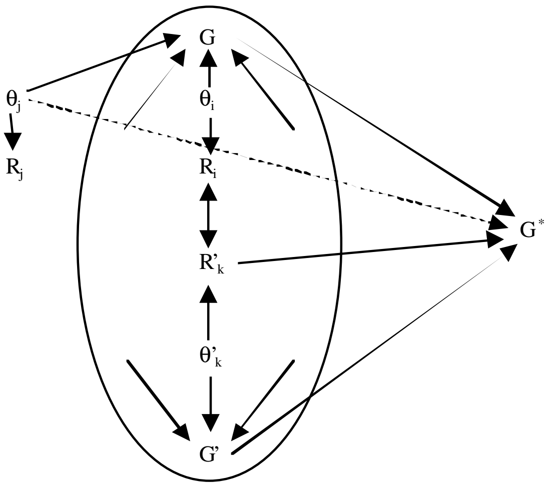
```
design inhibit-2 is
do never1: false → skip
[] never2: false → skip
```

Hence *ramification* can be seen as a primitive connector.

5.3 Role Overlay

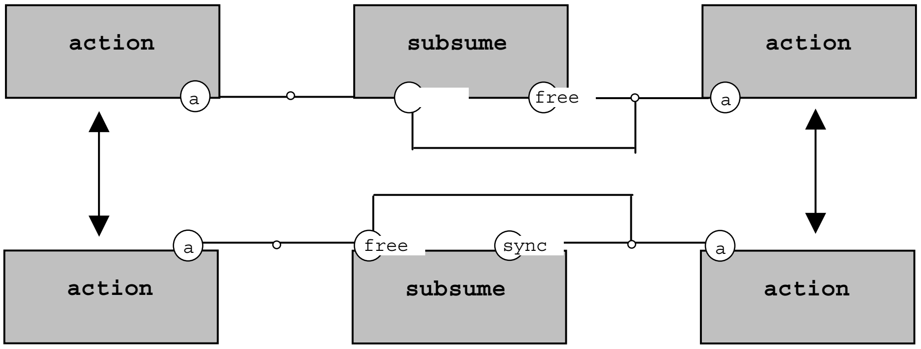
The third operation allows combining several connectors into a single one if they have some roles in common, i.e., if there is an isomorphism between those roles. The construction is as follows.

Consider an n-ary connector with roles R_i and glue G, and an m-ary connector with roles R'_k and glue G'. The glue of the new connector is calculated as follows. Consider the diagram that consists of all the connections that have isomorphic roles together with the isomorphisms, and calculate its colimit. The apex of the colimit is the new glue. Each of the pairs of connections involved gives rise to a connection of the new connector. The role of this connection is one of the roles of the old connections; because they are isomorphic, it does not matter which one is chosen (in the figure, we chose R'_k). This role is connected directly to the new glue through one of the morphisms that results from the colimit; hence, its cable is the role signature.



Each of the connections of the original connectors that is not involved in the calculation of the new glue, which means that it does not share its role with a connection of the other connector, becomes a connection of the new connector by composing the old new glue morphism with the colimit morphism that connects the old glue to the new one. This is exemplified in the figure with the connection with role R_j.

This operation provides a second way of showing that synchronisation is not a primitive connector in our catalogue. We can indeed obtain full synchronisation of two actions by making each one subsume the other. This is achieved by overlaying two copies of the subsumption connector in a symmetric way: the first (resp. second) role of one copy corresponds to the second (resp. first) role of the other copy. The diagram is



and its colimit makes all actions collapse into a single one. Hence the glue of the resulting connector is isomorphic to the *sync* component. Moreover, each pair of overlaid roles results into a single one, and therefore there will be only two *action* roles. In summary, the resulting connector is the synchronisation connector.

6 Concluding Remarks

In this paper, we have shown how (elementary) concepts of Category Theory can be used to formalise key notions of Software Architecture independently of the language chosen for describing the behaviour of components, thus providing a semantic domain for Software Architectures that complements the choice of a computational model. As illustrated by the proposed category of CommUnity designs, the categorical “style” favours the choice of formalisms in which the process of interconnecting components in configurations of complex systems is “separated” from the description of the internal computations that they perform. Therefore, it supports an approach to software development that is very close to what has been proposed in the area of Coordination Languages and Models [15].

The independence that was achieved relative to the choice of specific Architecture Desciption Language was taken one step further. We showed how the proposed mathematical framework supports generalisations of the notion of architectural connector and corresponding instantiation mechanisms that address important issues like stepwise development and compositional evolution of systems driven by architectures. The wealth of possibilities that this formal approach has opened to Software Architectures is being exploited in a “real-world” environment, namely through the extension of object-oriented and component-based modelling techniques. This effort is being pursued as a response from industrial needs for added flexibility in the way we structure our systems to facilitate the reuse of legacy systems and the dynamic reconfiguration of systems in a way that is compositional with the evolution of the business domains. The progress that has been made in this front is reported in a companion paper [3].

Further work is in progress at the more "theoretical level". These address the support that is required for an architecture-driven process of reconfiguration in which connectors can be added, deleted or replaced in run-time. Preliminary work in this direction is reported in [31,33]. We are also working on a notion of high-order connector through which we can integrate non-functional aspects like security and fault-tolerance into functional architectures. Work in this direction is reported in [24,32]. Finally, we intend to develop a logical calculus that supports the analysis of architectures, both in their static and dynamic aspects, including mechanisms for reasoning about possible interactions between connectors when applied simultaneously to a given system.

Acknowledgments

We would like to thank Luís Andrade from ATX Software SA for the opportunity to test and revise our theory against the practice of architecture building in the "real-world". We would also like to thank Roland Backhouse and Jeremy Gibbons for the opportunity to lecture this material at the Summer School on Generic Programming 2002, and the participants for much helpful feedback.

References

1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM,* 6(3), 1997, 213-249.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in R.France and B.Rumpe (eds), *UML'99 – Beyond the Standard*, LNCS 1723, Springer Verlag 1999, 566-583.
3. L.Andrade and J.L.Fiadeiro, "Service-Oriented Business and System Specification: Beyond Object-orientation", in H.Kilov and K. Baclwaski (eds), *Practical Foundations of Business and System Specifications*, Kluwer Academic Publishers 2003, 1-23.
4. L.F.Andrade, J.L.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger, "Patterns for Coordination", in *COORDINATION'00*, G.Catalin-Roman and A.Porto (eds), LNCS 1906, Springer-Verlag 2000, 317-322.
5. J.P.Banâtre and D.Le Métayer, "Programming by Multiset Transformation", *Communications ACM* 16(1), 1993, 55-77.
6. L.Bass, P.Clements and R.Kasman, *Software Architecture in Practice*, Addison Wesley 1998.
7. J.Bosch, "Superimposition: A Component Adaptation Technique", *Information and Software Technology* 1999.
8. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
9. J.L.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed), *Proc. 3rd Symp. on Foundations of Software Engineering*, ACM Press 1995, 72-80.

10. J.L.Fiadeiro and T.Maibaum, "A Mathematical Toolbox for the Software Architect", in J.Kramer and A.Wolf (eds), *Proc. 8th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1996, 46-55.
11. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming* 28, 1997, 111-138.
12. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
13. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in A.Haeberer (ed), *AMAST'98*, LNCS 1548, Springer-Verlag 1999.
14. N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
15. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35(2), 1992, 97-107.
16. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trappl (eds), *Advances in Cybernetics and Systems Research*, Transcripta Books 1973, 121-130.
17. J.Goguen, "Principles of Parametrised Programming", in Biggerstaff and Perlis (eds), *Software Reusability*, Addison-Wesley 1989, 159-225.
18. J.Goguen, "Parametrised Programming and Software Architecture", in *Symposium on Software Reusability*, IEEE 1996.
19. J.Goguen and R.Burstall, "Institutions: Abstract Model Theory for Specification and Programming", *Journal of the ACM* 39(1), 1992, 95-146.
20. S.Katz, "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15(2), 1993, 337-356.
21. A.Lopes, "Não-determinismo e Composicionalidade na Especificação de Sistemas Reactivos", PhD Thesis (in Portuguese), Universidade de Lisboa, Jan. 1999.
22. A.Lopes and J.L.Fiadeiro, "Using explicit state to describe architectures", in E. Astesiano (ed), *FASE'99*, LNCS 1577, Springer-Verlag 1999, 144–160.
23. A.Lopes and J.L.Fiadeiro, "Revising the Categorical Approach to Systems", in H.Kirchner and C.Ringeiseen (eds), *AMAST'02*, LNCS 2422, Springer-Verlag 2002, 426-440.
24. A.Lopes. M.Wermelinger and J.L.Fiadeiro, "Compositional Approach to Connector Construction", in M.Cerioli and G.Reggio (eds), *Recent Trends in Algebraic Development Techniques*, LNCS 2267, Springer-Verlag 2002, 201-220.
25. J.Magee, J.Kramer and M.Sloman, "Constructing Distributed Systems in Conic", *IEEE TOSE* 15 (6), 1989.
26. M.Moriconi and X.Qian, "Correctness and Composition of Software Architectures", in *Proc. Second Symposium on the Foundations of Software Engineering*, ACM Press 1994, 164-174.
27. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, 40-52.
28. V.Sassone, M.Nielsen and G.Winskel, "A Classification of Models for Concurrency", in E.Best (ed), *CONCUR'93*, LNCS 715, Springer-Verlag, 82-96.
29. M.Wermelinger and J.L.Fiadeiro, "Connectors for Mobile Programs", *IEEE Transactions on Software Engineering* 24(5), 1998, 331-341.
30. M.Wermelinger and J.L.Fiadeiro, "Towards an Algebra of Architectural Connectors: a Case Study on Synchronisation for Mobility", in *Proc. 9th International Workshop on Software Specification and Design*, IEEE Computer Society Press 1998, 135-142.
31. M.Wermelinger and J.L.Fiadeiro, "Algebraic Software Architecture Reconfiguration", in *Software Engineering – ESEC/FSE'99*, LNCS 1687, Springer-Verlag 1999, 393-409.
32. M.Wermelinger, A.Lopes and J.L.Fiadeiro, "Superposing Connectors", in *Proc. 10th International Workshop on Software Specification and Design*, IEEE Computer Society Press 2000, 87-94.
33. M.Wermelinger, A.Lopes and J.L.Fiadeiro, "A Graph Based Architectural (Re)configuration Language", in *ESEC/FSE'01*, V.Gruhn (ed), ACM Press 2001, 21-32.

Author Index