Generic Operations on Nested Datatypes

Ian Bayley

Balliol College

Thesis submitted for the degree of Doctor of Philosophy at the University of Oxford, Michaelmas 2001

Abstract

Nested datatypes are a generalisation of the class of regular datatypes, which includes familiar datatypes like trees and lists. They typically represent constraints on the values of regular datatypes and are therefore used to minimise the scope for programmer error.

An operation is said to be generic if it is parameterised by a datatype. This thesis explains how to define and reason with generic operations for nested datatypes. The operations we choose to illustrate the method include the zip and membership operations of Hoogendijk's thesis. These operations are thereby generalised from regular datatypes to nested datatypes.

We use fold operators to define and reason with these generic operations. It is therefore sufficient for us to define these fold operators generically and to express neatly generic theorems for these folds, such as universal properties and fusion laws. This we do for the three types of folds on nested datatypes. We demonstrate the theorems by proving the fold-equality law, which connects two varieties of fold representing two different modes of evaluation.

Much of our reasoning is with relations rather than functions so we have to adapt our semantics for nested datatypes to incorporate the operators of relational calculus. For this reason, we extend our fold operators and associated theorems to apply to relations.

Okasaki has argued informally that every nested datatype represents a constraint on some regular datatype. We prove this formally by defining an injective embedding function for each nested datatype. Since this operation connects programs that use nested datatypes with programs that use only regular datatypes, we can use it to remove nested datatypes from programs. There is some hope that we can also use it as a means of constructing programs for nested datatypes.

Acknowledgements

The research documented here was funded for three years by the Engineering and Physical Sciences Research Council (EPSRC).

The presentation of this thesis has been greatly improved by discussions with Richard Bird, who has kindly supervised me for the last four years, and by meetings with other members of the Algebra of Programming Group, especially Jeremy Gibbons and Clare Martin.

I received much vital support from Geraint Jones, my adviser, from Professor Henry McQuay, the Praefectus of Holywell Manor, and from Keith Hannabus and Joe Stoy, my College advisers. The staff at Comlab were also helpful, especially Jane Ellory, Julie Sheppard and Frances Page.

I have made many good friends through my interest in quizzing. I would like to acknowledge in particular Masterteam co-champions Brewis and Stainer, with whom I also founded the Quiz Society; Athar and Andrew, my Balliol and Oxford University team mates; and Roger for four years of enjoyable Monday evenings competing in his criminally underappreciated quiz.

Three years of my time in Oxford was spent living in Holywell Manor, the Graduate Centre of Balliol College. Among the many friends I made there, particular mention must be made of Brian, Stephen, Vicky and Elizabeth.

Draft copies of this thesis were reviewed by Ralf Hinze and Clare Martin. Oege de Moor and Geraint Jones examined my transfer report. Ralf and Oege also examined the thesis itself, corrected many errors and suggested many improvements. I have found all the people working in the small field of nested datatypes to be very approachable and helpful.

Finally, I would like to thank my mother, my father, my brother Stuart, my schoolfriends from Crosby, especially Arif and Daniel, and my girlfriend Jane, who has been remarkably caring and patient with me while I finish this thesis.

Contents

1 Introduction **1**

2 Simple Folds **13**

2.1 A naive category for Haskell programs . . . 13
2.2 A naive model for type constructors . . . 15
2.3 Non-recursive datatypes . . . 16
2.3.1 Products . . . 17
2.3.2 Bifunctors . . . 17
2.3.3 Coproducts . . . 18
2.3.4 Functors in general . . . 19
2.3.5 Natural transformations . . . 20
2.4 Regular datatypes . . . 21
2.4.1 Semantics of regular datatypes . . . 21
2.4.2 Standard folds on lists . . . 22
2.4.3 Example: summing a list . . . 23
2.4.4 Example: mapping a list . . . 24
2.4.5 Definition of regular functors . . . 24
2.4.6 Generic standard fold operator . . . 26
2.5 Nested datatypes . . . 27
2.5.1 Semantics of nested datatypes . . . 27
2.5.2 The endofunctor category . . . 27
2.5.3 Simple folds for nests . . . 28
2.5.4 Blampied's algebra family folds . . . 29
2.5.5 Example: Flattening a nest . . . 30
2.5.6 Existence of initial algebras . . . 31
2.5.7 Definition of nested functors . . . 32
2.5.8 Simple folds on alternating lists . . . 33
2.5.9 Example: separating an alternating list . . . 35

2.5.10 Generic simple fold operator 36
2.5.11 Simple folds for square matrices 36

3 Other folds for linear datatypes 38
3.1 Generalised folds for nests 39
3.1.1 Generalised fold operator for nests 39
3.1.2 Generalised fold operators in Haskell 40
3.1.3 Example: summing a nest 42
3.1.4 Reductions 44
3.1.5 Map-fusion laws 44
3.1.6 Fold-fusion laws 45
3.2 Efficient folds for nests 47
3.2.1 Improving on naive list reversal 48
3.2.2 Improving on generalised folds 48
3.2.3 Efficient summation on nests 49
3.2.4 Efficient folds in Haskell 51
3.3 Fold-equality law 51
3.3.1 Motivation for fold-equality law 51
3.3.2 Derivation of fold-equality law 52
3.3.3 Application: Flattening on nests 54
3.3.4 Comparisons of flattening functions 55
3.4 Folds for linear datatypes 56
3.4.1 Alternating lists 59
3.4.2 Square Matrices 59
3.5 Proofs of universal properties for folds 60
3.5.1 Uniqueness of generalised folds 61
3.5.2 Uniqueness of efficient folds 61
3.5.3 Proof sketch for the theorem 63
3.6 Map-fusion law for efficient folds 63

4 Other folds for non-linear datatypes 65
4.1 Generalised folds for bushes 67
4.2 Alternative grammar for polynomial hofunctors 69
4.3 Generic generalised fold operator 70
4.4 Generic summation 71
4.5 Generic map-fusion law for generalised folds 73
4.6 Generic fold-fusion law for generalised folds 75
4.7 Efficient folds for non-linear datatypes 78

4.8 Map-fusion law for efficient folds 80

5 Relators **81**

5.1 The category of relations 82
5.1.1 Relational product 84
5.2 A category of lax natural transformations 85
5.2.1 Relators 85
5.2.2 Lax natural transformations 86
5.2.3 Further structure of the endorelator category 87
5.3 Nested functors extend to relators 87
5.3.1 Power allegories 88
5.3.2 Endofunctor categories based on power allegories . . . 89
5.3.3 Initial algebras extend to endorelator category 91
5.3.4 Efficient fold operators preserve total functions 92
5.3.5 Map operations are efficient folds 92
5.3.6 The functor Nest commutes with converse 92
5.3.7 All nested functors commute with converse 93
5.3.8 Nested relators are monotonic 95
5.3.9 Generalising to other allegories 95
5.4 Relational fusion laws 96
5.4.1 Knaster-Tarski theorem 96
5.4.2 Relational fold-fusion law 97
5.4.3 Relational map-fusion law 99
5.5 Hylomorphism theorem 100
5.6 Fold operators preserve injectivity 100
5.6.1 Left-division 100
5.6.2 Proof that fold operators preserve injectivity 101
5.7 Fold-fusion law for efficient folds 103

6 Membership **106**

6.1 Introduction 106
6.2 Characterisation 108
6.3 Candidate membership 109
6.3.1 Candidate membership: polynomial cases 109
6.3.2 Candidate membership: fixpoint case 111
6.4 Candidate membership is membership 113
6.5 Fans 114
6.5.1 Fans for polynomial cases 114

6.5.2 Unfans for nests 115
6.5.3 Fans for fixpoint case 116
6.6 Generalised unfans are efficient folds 116

7 Zips **120**
7.1 Introduction 120
7.1.1 An example of a zip 120
7.1.2 Another example of a zip 121
7.1.3 Informal specification 122
7.1.4 Applications 122
7.1.5 Multirelators 123
7.1.6 Chapter overview 123
7.2 Inductive definition of candidate zip 124
7.2.1 Candidate zips for polynomial endorelators 124
7.2.2 Candidate zips for polynomial binary relators 125
7.2.3 Candidate zips for nested fixpoints 127
7.2.4 Illustration of candidate zip 128
7.3 Generic properties of zips 130
7.3.1 Formalising shape behaviour 130
7.3.2 Alternative requirements 130
7.3.3 Higher-order naturality 131
7.3.4 Hoogendijk's theorem of zips 132
7.4 Proofs of properties 133
7.4.1 Zips are higher-order natural 133
7.4.2 Zips are compositional 134
7.4.3 Zips respect identities 136
7.4.4 Linear nested relators are almost commuting 136
7.4.5 Zips are natural transformations 137
7.5 From endorelators to multirelators 138

8 Embedding Functions **140**
8.1 Introduction 140
8.2 Embedding target for nests 142
8.3 Embedding function for nests 144
8.4 Embedding bushes 144
8.5 Embedding arbitrary datatypes 145
8.6 Fold-fusion law for embedding operator 147
8.7 Fold-equivalence law 148

8.7.1 Embedding functions preserve flattening 150
8.7.2 Example: summation 150
8.7.3 Example: unfanning 151
8.7.4 Example: last 151
8.8 Another law for efficient reductions 153
8.9 Laws for arbitrary combinators 155
8.10 Nested relators have membership 158
8.11 Embedding predicates 159

9 Conclusion **161**

Bibliography **172**

Chapter 1

Introduction

Consider the following Haskell datatype *Tree*, whose instances are leaf-labelled binary trees.

$$\begin{array}{lcl}\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Pair}\ (\mathit{Tree}\ a)) \\ \textbf{type}\ \mathit{Pair}\ a & = & (a, a)\end{array}$$

We say that *Tree* is a *regular* datatype because the recursive use of *Tree* takes the same parameters as the defining use. Almost all functional programming is conducted exclusively with the class of regular datatypes, which includes other kinds of trees, and lists as well. *Nested* datatypes are a generalisation of regular datatypes where the above restriction on parameters is removed. For example, the datatype *Pow* below is a nested datatype, though not a regular datatype, because its recursive use has the parameter *Pair a*, whereas its defining use has parameter a.

$$\textbf{data}\ \mathit{Pow}\ a = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Pow}\ (\mathit{Pair}\ a))$$

Note that the only difference between *Pow* and *Tree* is in the position of the type constructor *Pair*. An instance of *Pow* is either a single element or a pair of elements or a pair of pairs of elements, and so on. If we ignore the constructor functions then every instance of *Pow* matches an instance of *Tree*, so *Pow* represents a subtype of *Tree*. The members of the subtype are those instances of *Tree* that are complete, that is, perfectly balanced. We shall call the members of the subtype *power trees* since the number of leaves they contain is a power of two, though elsewhere [BGJ00] the term power list is preferred as the tree structure is not there considered important.

The datatype *Pow* comes in handy if we want to write a function that returns a complete leaf-labelled tree [Hin00a]. Such a structure is often required by paralell algorithms. Ross Paterson gives in [Pat01] an interesting variant: a datatype of lists of depth-preserving functions on trees that are ascending in depth. This is also useful for parallel algorithms. If we use *Pow* instead of *Tree*, then we cannot by mistake write a correctly-typed function that returns an unbalanced tree. In other words, we can use the Haskell type checker to automatically check that our function has a certain property and we do this by using a more informative datatype that conveys the extra information that the output must be perfectly balanced. Many more of these more informative types, including other kinds of balanced tree and matrices, can be phrased as nested datatypes.

In fact, we shall discover in Chapter 8 that every nested datatype represents a subtype of some regular datatype. The conclusion to draw from this is that nested datatypes do not add to the expressivity of the Haskell language. However, they are still useful because they act as "regular datatypes plus constraints". Nevertheless, Okasaki demonstrates with the example of the trie data structure [Oka98b] that nested datatypes can be designed without having to keep in mind a particular regular datatype.

Programming with nested datatypes is made much simpler by the use of fold operators, an idea that originated with Bird's theory of lists [Bir89]. A great number of recursive functions are folds, that is, they fit a particular pattern of recursion. We encapsulate this pattern in a fold operator that returns a fold operation given certain ingredients that are supplied as parameters. The operator can be used to write fold operations without using recursion explicitly. The Standard Prelude implements the fold operator for built-in lists as the function *foldr*, illustrated by

$$\mathit{foldr}\, f\ e\, [1,2,3] \quad = \quad f\, 1\ (f\, 2\ (f\, 3\ \ e))$$

Many folds are also maps so we define, as a special case of the fold operator, a map operator to produce maps. The map operator for built-in lists is implemented in the Standard Prelude as the function *map*, illustrated by

$$\mathit{map}\, f\, [1,2,3] \quad = \quad [f\, 1, f\, 2, f\, 3]$$

It is easy to extend the notion of folds to any regular datatype by observing

$$[1,2,3] \ = \ (:)\, 1\ ((:)\, 2\ ((:)\, 3\, [\,])) \ = \ \mathit{foldr}\ (:)\, [\,]\ [1,2,3]$$

It is clear that *foldr f e* simply replaces the constructor functions (:) and [ ] with the parameters f and e. In general, the fold operator for a datatype uses its parameters to replace all occurrences of constructor functions in a structure. We can write a fold operator for *Pow* that does exactly this but we shall see in the next chapter that the type of such an operator must be so restricted that it cannot be used for much. We call it a simple fold operator to distinguish it from the standard fold operators that are defined for regular datatypes. In [BP99b], Bird and Paterson introduced a more useful generalised fold operator and in [Hin99a], Hinze introduced an efficient fold operator, which is yet more general and can be used to produce map operations.

Fold operators are invaluable for program calculation, the process of transforming an obviously correct but inefficient specification into a less clear but more efficient implementation. Two theorems for folds that aid this activity are the map-fusion law and the fold-fusion law. The map-fusion law for lists rewrites the expression $\mathit{foldr}\, f\ e \cdot \mathit{map}\, k$ as a single fold. The fold-fusion law for lists rewrites the expression $k \cdot \mathit{foldr}\, f\ e$ as a single fold, but only if certain conditions are met.

Malcolm showed [Mal89] that every regular datatype has both a fold-fusion law and a map-fusion law associated with its standard fold. Bird and Paterson derived map-fusion and fold-fusion laws for their generalised fold operator in [BP99b]. Their starting point was the so-called universal property of generalised folds, which both defines the generalised fold operator and asserts that generalised folds are uniquely defined by this property.

Often program calculation starts from a non-deterministic specification so we must reason with relations instead of functions. This means we have to extend to nested datatypes the previous work of Oege de Moor [BdM97] and Roland Backhouse [BJJM99] designed to enable reasoning with relations for regular datatypes. Some thought is needed to confirm that fold operators and map operators can be extended to relations.

The main purpose of this thesis is to define and prove properties of *generic* operations. These are operations that are parameterised by a datatype. As a first example, Hoogendijk [Hoo97] generalises the *unzip* function of Haskell to a generic operation *zip* that takes as a parameter a pair of regular datatypes.

We extend this further to nested datatypes. We define *zip* using a generic generalised fold operator and we prove that it obeys certain generic properties of zips by using generic versions of the universal property and fusion laws for generalised folds.

In fact, *zip* is only defined for so-called linear datatypes. A datatype is *linear* if the parameter to all recursive uses in its definition is not modified by the datatype being defined. Although *Pow* is a linear datatype, the datatype of bushes, for example, is not so we say that it is *non-linear*.

$$\mathbf{data}\ \mathit{Bush}\ a \quad = \quad \mathit{NilB} \mid \mathit{ConsB}\ (a, \mathit{Bush}\ (\mathit{Bush}\ a))$$

Although Bird and Paterson define the universal property and fusion laws for generalised folds on non-linear nested datatypes, we find that we must rewrite them in a special way that facilitates generic reasoning.

Now we can list the major contributions of the thesis.

- we define a zip operation for each linear nested datatype and prove that it has the properties expected of zips
- we explain how to reason generically with the universal property and fusion laws of generalised folds
- we prove the fold-equality law for linear datatypes. This law connects two sorts of folds — simple folds and efficient reductions — by giving conditions when they are equal. The two folds represent two different forms of evaluation.
- we define and verify a generic membership relation, thereby confirming that nested datatypes have membership, an essential property of datatypes [HdM00]
- we confirm that our reasoning with nested datatypes can be performed with relations instead of functions
- we define a function that embeds every nested datatype within a regular datatype, explain how to use it to remove nested datatypes from programs, and give pairs of programs that are equivalent according to the embedding

The layout of this thesis is as follows.

Chapter 2 gives a semantics for regular datatypes based on the notion of standard folds. An important mathematical structure known as a category is used to describe the semantics. Using categories allows us to reason generically. Type constructors are represented by functors, which are mappings on categories. A similar semantics is given for nested datatypes. Since a different category is needed for this, the associated folds are called simple folds.

Chapter 3 explains that simple folds cannot be used to perform summation operations so it introduces generalised folds and gives their fusion laws. These are generalised further to efficient folds, which include map operations. The operators for both of these folds are defined by universal properties that, in the interests of concise reasoning, have no case analysis. After deriving the fold-equality law, we generalise the chapter to linear datatypes and prove the universal properties of both types of fold operators.

Chapter 4 generalises Chapter 3 to non-linear datatypes, by expressing the functor equations that define datatypes in a special form. Generic generalised and efficient operators are given and a generic sum operation is also given in order to demonstrate how the generic fold operators can be used to define generic operations. To enable us to prove properties of these operations, we derive convenient forms of the generic fold-fusion and map-fusion laws.

Chapter 5 explains the need to reason with relations and changes the category to one that can be augmented with the operators of the relational calculus. Then for many of the concepts explained in the last three chapters, we introduce relational extensions including relational products, lax natural transformations and relators. The fusion laws are restated with inequalities.

Chapter 6 defines a membership relation for each nested datatype and proves that it satisfies the characterisation of membership. Hence nested relators have membership; this is an important assurance that nested relators correspond to datatypes. The proof of this is an application of the map-fusion law of efficient folds.

Chapter 7 defines a zip operation for each linear nested datatype, thereby

extending [Hoo97], and proves that the zip operations defined meet the requirements of zips. The definition and proofs use generalised folds where Hoogendijk [Hoo97] uses standard folds.

Chapter 8 motivates Chris Okasaki's rules [Oka98a] for finding a regular supertype for any nested datatype and defines an embedding function that embeds the nested datatype in a regular datatype. This function describes a simulation between programs that use nested datatypes and programs that use regular datatypes. The chapter then gives rules for deriving either of these from the other.

Chapter 9 discusses how effective our system is for generic reasoning and what we have learned about the usefulness of the universal properties and fusion laws of our fold operators, and gives a few other insights. It then proposes future work.

Finally, let us conclude this introduction by focusing on two important areas of background. Shortly we shall summarise the support that Haskell gives for nested datatypes but first we shall try to get some idea of how expressive nested datatypes are.

One simple way of designing nested datatypes is to construct an invariant for its parameters to satisfy as the datatype recurses. For example, the parameter of *Pow* always represents a perfectly balanced tree. Now we shall use an invariant to construct Ross Paterson's datatype of AVL trees [Pat98]. Suppose that l represents an AVL tree of height k and h represents an AVL tree of height $k+1$. Then an AVL tree of height $k+2$ is given by $AVLNode\ a\ l\ h$ where a is the type of the elements contained in both l and h, and $AVLNode$ is defined by

$$
\begin{aligned}
\mathbf{data}\ AVLNode\ a\ l\ h \quad &= \quad LeftLarger\ h\ a\ l \\
&\mid \quad EqualHeight\ h\ a\ h \\
&\mid \quad RightLarger\ l\ a\ h
\end{aligned}
$$

So if $k = 0$ in the above, then $AVL'\ a\ l\ h$ gives every possible AVL tree, for AVL' defined by

$$
\begin{aligned}
\mathbf{data}\ AVL'\ a\ l\ h \quad &= \quad AVLZero\ l \\
&\mid \quad AVLSucc\,(AVL'\ a\ h\,(AVLNode\ a\ l\ h))
\end{aligned}
$$

Observe that the third parameter represents a tree of height one greater than the second both before and after recursion. This is our invariant. We therefore make our type of AVL trees be AVL below, thereby ensuring that $k = 0$ when AVL' is first used.

$$\textbf{type}\ AVL\,a \quad = \quad AVL'\,a\,()\,a$$

Our final example is Ross Paterson's datatype of square matrices [Pat98], a variant of which is given in [Oka99]. The datatype we develop is higher-order, that is, it is parameterised by a type constructor. The invariant we shall now use is that the type constructor parameter f is equal to $Cons^k\ Nil$, for some k, where $Cons$ and Nil are defined by

$$\begin{array}{lcl} \textbf{data}\ Nil\ a & = & Nil \\ \textbf{data}\ Cons\,f\ a & = & Cons\ a\ (f\ a) \end{array}$$

Then $f\ a$ is a list, with length k, of a's and $f\ (f\ a)$ is a list with length k of lists of a's, each of which have length k. This is the same as a $k \times k$ square matrix of a's. So if $k = 0$ then the datatype SM below generates every possible square matrix.

$$\begin{array}{lcl} \textbf{data}\ SM\,f\ a & = & ZeroSM\ (f\ (f\ a)) \\ & | & SuccSM\ (SM\ (Cons\,f)\ a) \end{array}$$

Now we ensure that $k = 0$ by defining the type of square matrices to be

$$\textbf{type}\ Square\ a \quad = \quad SM\ Nil\ a$$

These two examples give us some idea of the constraints on regular datatypes that nested datatypes can capture. For a start, the constraints are on shape rather than value. We shall define shape formally in chapter 7 but for the moment, an informal intuition suffices: the shape of a data structure is what stays the same when its contents are changed.

We can represent each possible shape by a natural number because the essential property of any shape is its size. Therefore, we can represent all the possible shapes of a datatype by a bag of natural numbers. This is what Hinze suggests we do in [Hin01]. For example, the datatype of lists can be represented by a bag that contains each natural number exactly once because the datatype has one shape for each possible length. There is a simple syntactic translation from recursive bag equations to Haskell datatypes.

Kevin and Roland Backhouse [BB00] suggest a different approach to that of Hinze, whereby nested datatypes are written in Haskell instead of the intermediate language of bag equations. However, the size numbers are made explicit again, by using type constructors to represent Church numerals. Each new shape produced by the inductive case of the definition can be passed to a Church-encoded predicate and either kept or discarded according to the result of testing it with the predicate.

Since any computable predicate can be Church-encoded, it would seem possible to design a datatype of, for example, lists having prime number length. Unfortunately, this important test case could not be realised because it required that type constructors be polymorphic in kind. Kinds [McC79] are a type of types. The kind of a value records whether the value is a type or a type constructor and if it is the latter, the kinds of each of its arguments.

These suggestions from Hinze and the Backhouses can be used to design nested datatypes. Unfortunately, neither describe nested datatypes as constraints on regular datatypes as appears to be helpful. Borges' suggestion [Bor01] differs in both these respects. He shows how to define for each linear nested datatype a predicate on the underlying regular datatype. This predicate can be thought of as encapsulating the meaning of the nested datatype.

Using nested datatypes is one of many ways to make a type checker verify properties of our programs. Here we shall briefly review the alternatives and their applications. One approach is that of refinement types described in [FP91] for the language ML. Refinement types are subtypes of regular datatypes and they form a lattice with intersection and union of types as meet and join respectively. The type inference algorithm works by performing abstract interpretation on this lattice, which the programmer specifies by describing how the constructor functions of the regular datatype map refinement types to refinement types. Examples in [FP91] and [Pie91] include lambda-terms in head normal form, Church booleans and numbers (with associated arithmetic operations) and bit strings that have no leading zeros (with addition defined).

There is more overlap in examples with dependent types. A function is dependently-typed if its type depends on the value of one of its parameters.

Dependent ML (DML) [Xi98], implemented by Hongwei Xi, has a limited facility for dependent types in that it uses particular constraint domains to constrain type signatures. The examples given in [Xi98, XP99] of functions with DML-checkable properties are quicksort, list concatenation and binary search. The last of these also illustrates some extra advantages in automated optimisation and dead code elimination.

Furthermore, Xi has implemented in DML [Xi99] Braun trees, random-access lists, binomial heaps and red-black trees, all of which can also be captured as nested datatypes [Oka98b]. He does this by taking a regular datatype, using a natural number to index some of the possible shapes and then using the numbers to impose certain constraints on the substructures at each node.

For example, we can define a type similar to that of power trees by constraining the type of leaf-labelled trees so that for every internal node the two immediate subtrees have the same height.

$$\begin{array}{l}\textbf{datatype}\ 'a\,pow\ \textbf{with nat}\ = \\ tip(0)\ |\ \{n : \textbf{nat}\}\ bin(n+1)\ \textbf{of}\ \ 'a\,pow(n) *\ 'a\,pow(n)\ ;;\end{array}$$

Now $'a\,pow(n)$ denotes the type of perfectly balanced leaf-labelled binary trees whose elements have type $'a$. The constructor function *bin* has type

$$'a\,pow(n) * \,'a\,pow(n) \rightarrow pow(n+1)$$

This means that *bin* takes a pair of perfectly balanced trees with equal height and joins them to give a perfectly balanced tree with height one greater. This is a very precise type. The nearest we can come to it in Haskell with nested datatypes is the following:

$$\begin{array}{lcl} bin & :: & (Pow\ a, Pow\ a) \rightarrow Pow\ a \\ bin\ (Zero\,x, Zero\,y) & = & Succ\ (Zero\ (x,y)) \\ bin\ (Succ\,x, Succ\,y) & = & Succ\ (bin\ (x,y)) \end{array}$$

The type signature given to this version of *bin* is far less precise than that of its counterpart in DML. Although the result is required to be perfectly balanced, the input trees need not have the same height and the output need not be higher by exactly one. This illustrates a weakness of nested datatypes: when specifying the desired properties of a function, we can constrain the result but we cannot state a relationship between the arguments and the

result of a function. Consequently, *bin* in Haskell is partial so it will give a run-time error if applied to the pair $(\mathit{Zero}\,1, \mathit{Succ}\ (\mathit{Zero}\ (2,3))$. In DML, however, this error would be caught during compilation.

Even without the more accurate type signatures, DML still appears to be superior because DML functions for power trees can easily be obtained from functions for trees, simply by adding an appropriate type signature, whereas for nested datatypes, the conversion is considerably more complicated. The two *bin* functions illustrate this but to reinforce the point let us consider another way of combining two perfectly balanced trees. The following DML code zips a pair of trees of height n to give a tree of height $n+1$.

$$
\begin{array}{l}
\textbf{fun}\ \mathit{ziptr}\ (\mathit{tip}(x), \mathit{tip}(y)) = \mathit{bin}(\mathit{tip}(x), \mathit{tip}(y)) \\
\mid \mathit{ziptr}\ (\mathit{bin}(x,x'), \mathit{bin}(y,y')) = \mathit{bin}(\mathit{ziptr}\ (x,y), \mathit{ziptr}\ (x',y')) \\
\quad \textbf{withtype}\{n : \textbf{nat}\}\langle n\rangle \Rightarrow\ '\!a\, \mathit{pow}(n) *\ '\!a\, \mathit{pow}(n) \rightarrow\ '\!a\, \mathit{pow}(n+1)
\end{array}
$$

This function is very much more difficult to write in Haskell. For a start, we need map operator *pow* for the type of power trees; we shall explain how to derive *pow* in Chapter 3.

$$
\begin{array}{lcl}
\mathit{pair} & :: & (a \rightarrow b) \rightarrow \mathit{Pair}\ a \rightarrow \mathit{Pair}\ b \\
\mathit{pair}\, f\ (x,y) & = & (f\ x, f\ y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{pow} & :: & (a \rightarrow b) \rightarrow \mathit{Pow}\ a \rightarrow \mathit{Pow}\ b \\
\mathit{pow}\ f\ (\mathit{Zero}\ a) & = & \mathit{Zero}\ (f\ a) \\
\mathit{pow}\ f\ (\mathit{Succ}\, x) & = & \mathit{Succ}\ (\mathit{pow}\ (\mathit{pair}\, f)\ x)
\end{array}
$$

Now we can zip two power trees together with *ziptr*, defined below.

$$
\begin{array}{lcl}
\mathit{shuffle} & :: & \mathit{Pair}\ (\mathit{Pair}\ a) \rightarrow \mathit{Pair}\ (\mathit{Pair}\ a) \\
\mathit{shuffle}\ ((a,b),(c,d)) & = & ((a,c),(b,d))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{zipow} & :: & \mathit{Pair}\ (\mathit{Pow}\ a) \rightarrow \mathit{Pow}\ (\mathit{Pair}\ a) \\
\mathit{zipow}\ (\mathit{Zero}\ a, \mathit{Zero}\ b) & = & \mathit{Zero}\,(a,b) \\
\mathit{zipow}\ (\mathit{Succ}\, x, \mathit{Succ}\, y) & = & \mathit{Succ}\ (\mathit{pow}\ \mathit{shuffle}\ (\mathit{zipow}\ (x,y)))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ziptr} & :: & \mathit{Pair}\ (\mathit{Pow}\ a) \rightarrow \mathit{Pow}\ a \\
\mathit{ziptr} & = & \mathit{Succ} \cdot \mathit{zipow}
\end{array}
$$

Before moving on, let us note some weaknesses of DML. Not all functions have a principal type, that is, a unique most general type. Also, power trees

in Haskell have far fewer constructors to pack and unpack, an important efficiency concern. Finally, it appears that there are some nested datatypes, including all the non-linear datatypes, that cannot be expressed using DML. There do not appear to be any advantages in combining DML and nested datatypes in a new language.

A fuller implementation of dependent types is offered by Cayenne [Aug98] but type checking is undecidable for general recursion, although checking will still terminate for structured recursion. Finally, indexed types [Zen97] offer similar possibilities to those of DML. Lists can be labelled with their lengths and used to construct arrays that are labelled with their dimensions. Then matrix multiplication, for example, can be given a type signature dictating that two matrices can only be multiplied together when their dimensions fit.

Now we shall examine the support that Haskell gives for nested datatypes. The following function, which computes the height of a power tree, will be accepted by a Haskell compiler.

$$\begin{array}{lcl} \mathit{height} & :: & \mathit{Pow}\ a \rightarrow \mathit{Int} \\ \mathit{height}\ (\mathit{Zero}\ x) & = & 0 \\ \mathit{height}\ (\mathit{Succ}\ y) & = & 1 + \mathit{height}\ y \end{array}$$

Unusually, the type signature is not optional. This is because the function features *polymorphic recursion*, that is, it calls itself with a type different from that at which it is defined. Clearly, any function defined by recursion on a datatype that is not regular has this property. Polymorphic recursion was forbidden by version 1.3 of Haskell but it is allowed by version 1.4. This is because the two versions are based on two slightly different type systems: Hindley-Milner [Mil78] and Mycroft-Milner [Myc84] respectively. The latter has a more liberal fixpoint case that allows recursive calls, like the one featured above, that have a more specific type than the original call.

However, type inference is undecidable for Mycroft-Milner [Hen93] and that is why the programmer must provide a type signature. (Note that type signatures are also compulsory for DML but not for the implementation of refinement types in ML mentioned earlier.) There is some discussion of the importance of this undecidability result in [Hen93], where the author suggests that, like the theoretical intractability of Hindley-Milner, it may not show up in practice.

Some higher-order functions, like fold operators for example, require rank two type signatures [JL], which we shall introduce in the next chapter. This syntax is a new feature of the Haskell standard that was originally introduced for Hugs 1.3C by Mark Jones, who described the associated typing rules of System F in [Jon97].

Since we define generic operations in this thesis, it would be useful to have support for them in Haskell. In particular, programs that make extensive use of libraries of generic operations will require less rewriting when the datatypes are changed than programs that do not. Application areas where this facility is useful include parsing, pretty printing and unification.

The language PolyP [JJ97] had a construct for defining generic operations but its genericity was limited to first-order regular datatypes that have exactly one parameter and no mutual recursion. Ralf Hinze's solution [Hin00d] to this problem was to use the kind of the datatype to index all the possible types that the operation can have. He also reduced the number of cases needed to specify a generic operation. These ideas have been implemented as the language Generic Haskell [Jeu01].

Chapter 2

Simple Folds

Let us motivate a calculus for calculating and reasoning about Haskell programs. First of all, how should we model types? Treating them as sets of values is simple enough. Then the type *Int*, for example, would be modelled by the set of integers. Haskell functions would then be modelled by functions on sets. However, this view is too simplistic and naive because functions can fail to terminate. Nevertheless, if we restrict our attention to those Haskell functions that always terminate, then the naive view is of great help in motivating the basics of our calculus.

2.1 A naive category for Haskell programs

We begin by noting three facts about Haskell.

- each function has both a source type and a target type, and
- any two functions can be composed, using an associative operator, provided the source of one matches the target of the other, and
- every type has defined on it an identity function, which leaves the input unchanged and acts as a unit of composition.

These three facts echo the three axioms that define a category. Formally, a *category* is a class of objects together with a class of arrows such that

- every arrow has both a *source* object and a *target* object; we write $f : X \to Y$ to indicate that arrow f has source X and target Y, and

- for every pair of arrows with matching source and target, $f : Y \to Z$ and $g : X \to Y$, the *composition* $f \cdot g : X \to Z$ exists. We have

  $$(f \cdot g) \cdot h \quad = \quad f \cdot (g \cdot h)$$

  for any further arrow, $h : W \to X$, and

- for each object X, there is an *identity* arrow $id_X : X \to X$. These arrows satisfy for each $f : Y \to Z$

  $$f \cdot id_Y = f = id_Z \cdot f$$

These axioms are so elementary that we shall use them frequently without comment. Our plan, once more, is to represent types as sets and always-terminating Haskell functions as total functions. The universe of all always-terminating Haskell functions is then represented by a category **Fun**, which has sets for objects and total functions for arrows. Functions are now defined using composition rather than application. These two styles for defining functions are respectively named *point-free* and *pointwise*. The advantage of point-free proofs is that there is no case analysis on the inputs to functions. Such proofs can therefore easily be made generic.

Again, we must emphasize just how naive is the way in which we model the Haskell language. The category we should ideally use is $\mathbf{CPO}_\perp$, which has pointed complete partial orders as objects and strict continuous functions as arrows. There are two major ways in which our naivete can cause us problems. First of all, some Haskell functions are partial, but we shall introduce in Chapter 5 a category **Rel** some arrows of which are partial functions. In the meantime, we shall only derive Haskell programs that always terminate. The other flaw is that we cannot deal with infinite data structures. Fortunately, we have no need to consider algorithms that produce or consume such structures in this thesis. Nevertheless, we shall explain when we come to them, how the universal properties of coproducts and standard folds must be modified if our choice of category were to change from **Fun** to $\mathbf{CPO}_\perp$.

Section 2.2 explains how in our naive view type constructors in Haskell correspond to functions on sets. Sections 2.3, 2.4 and 2.5 apply this intuition to non-recursive, regular and nested datatypes respectively. Standard folds and simple folds arise from the semantics of regular and nested datatypes respectively. Good introductions to the material in this chapter include [BJJM99]

and [BdM97]. Neither of these cover Section 2.5, however, which is based on [BM98] and [BP99b].

2.2 A naive model for type constructors

Let us examine a typical recursive datatype.

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

In words this declares that "a list is empty *or* an element *and* a list". The word *or* translates case selection. The word *and* translates tupling of types.

Now *List* is a type constructor, that is, a function from types to types. So there must be a corresponding function *List* on sets defined as follows: if A is a set then $\mathit{List}\ A$ is the set of all the lists that can be formed from the members of A. In order to define the set $\mathit{List}\ A$ formally, we declare that there is a bijection between it and some set formed from A and $\mathit{List}\ A$.

$$\mathit{List}\ A \quad \approx \quad \mathit{Base}\ (A, \mathit{List}\ A)$$

Here, *Base* is a binary operator on sets that we shall construct in the next section. This is an example of a fixpoint equation; we have $T \approx F\ T$ where $T = \mathit{List}\ A$ and $F\ X = \mathit{Base}\ (A, X)$. The symbol $\approx$ indicates an isomorphism between objects of a category: an *isomorphism* between objects A and B of an arbitrary category is a pair of arrows $f : A \to B$ and $g : B \to A$ such that $f \cdot g = id_B$ and $g \cdot f = id_A$. If this condition holds then we say that A and B are *isomorphic*. If the category is **Fun** then the isomorphisms are bijections.

Any regular datatype T can be put in the same form as *List* by using a different binary operator instead of *Base*. This abstraction will be useful later, because it means that we can define operations on regular datatypes in a generic fashion.

We shall overload the identifier *List* by making it also denote a map operator, which takes functions to functions. In Haskell, the map operator corresponds to the function *map* on built-in lists. Rewritten for our datatype *List*, that

function is

$$
\begin{array}{lcl}
\mathit{list} & :: & (a \to b) \to (\mathit{List}\ a \to \mathit{List}\ b) \\
\mathit{list}\, f\ \mathit{Nil} & = & \mathit{Nil} \\
\mathit{list}\, f\ (\mathit{Cons}\ (a, x)) & = & \mathit{Cons}\ (f\ a, \mathit{list}\, f\ x)
\end{array}
$$

(The syntax of Haskell forces us to write the map operator *List* with a small 'l'.) Now, *List* is what we call an endofunctor on **Fun**. An *endofunctor* on a category **C** is a mapping F between the objects of **C** together with a mapping also denoted F between the arrows of **C** such that if $f : A \to B$ then

$$F\, f : F\ A \to F\ B$$

It must also preserve identities and composition, that is,

$$
\begin{array}{rcl}
F\ id_A & = & id_{F\ A} \\
F\ (f \cdot g) & = & F\, f \cdot F\ g
\end{array}
$$

2.3 Non-recursive datatypes

Now we define the binary operator *Base*. By examining the Haskell definition of *List*, we realise that *Base* corresponds to the following datatype:

$$\mathbf{data}\ \mathit{Base}\ a\ b = \mathit{Nil}\ (\,)\ |\ \mathit{Cons}\ (a, b)$$

Here we have suggested that *Nil*, being a constant, can also be seen as a nullary operator. That is why we have applied *Nil* to the type ( ). This type has one value, also denoted ( ), and is represented in **Fun** by a one-element set. Such sets are terminal objects of **Fun** as the following definition makes clear. A *terminal object* is an object T with the property that for every object A, there is exactly one arrow, denoted $!_A$, from A to T. Any two terminal objects are isomorphic. Therefore, we shall pick one, call it *the* terminal object and denote it by $\mathsf{1}$. Here is *Base* as a binary operator on sets.

$$\mathit{Base}\ (X, Y) \;=\; \mathsf{1} + X \times Y$$

The infix operators $+$ and $\times$ are defined by

$$
\begin{array}{rcl}
A + B & = & \{\mathit{inl}\ a \mid a \in A\} \cup \{\mathit{inr}\ b \mid b \in B\} \\
A \times B & = & \{(a, b) \mid a \in A, b \in B\}
\end{array}
$$

The total functions $inl : A \to A+B$ and $inr : B \to A+B$ tag the elements of A and B, ensuring that for all x, $inl\, x \neq inr\, x$. Without the tagging, $A + B$ would be a union type [FP91]. Now we see that $\times$ is tupling and $+$ represents selection between cases. More abstractly, $\times$ is the product operator for the category **Fun** and $+$ is the coproduct operator. We shall now define these constructions for an arbitrary category.

2.3.1 Products

The *product* of the objects A and B is an object $A \times B$ and two arrows, $outl_{A,B} : A \times B \to A$ and $outr_{A,B} : A \times B \to B$. For any further pair of arrows $f : C \to A$ and $g : C \to B$, there should be a unique third arrow $\langle f, g\rangle : C \to A \times B$. This arrow is called the *fork* of f and g, and it is defined as follows. For any $h : C \to A \times B$,

$$h = \langle f, g\rangle \quad \equiv \quad outl_{A,B} \cdot h = f \quad \text{and} \quad outr_{A,B} \cdot h = g$$

This definition is an example of a *universal property*: an arrow $\langle f, g\rangle$ is defined by giving a property, the right-hand side of the equivalence, that only it satisfies.

In Haskell, the product of two types a and b is the tuple (a, b) together with the projection functions *fst* and *snd*, defined by $fst\ (x, y) = x$ and $snd\ (x, y) = y$. We implement the fork operator by

$$\begin{array}{lcl} fork & :: & (c \to a, c \to b) \to c \to (a, b) \\ fork\ (f, g)\, x & = & (f\, x, g\, x) \end{array}$$

The function $fork\ (f, g)$ duplicates its input, applies f to one copy and g to the other.

2.3.2 Bifunctors

We define $\times$ to be a binary operator on arrows too.

$$f \times g \quad = \quad \langle f \cdot outl, g \cdot outr\rangle$$

The action of $\times$ on arrows is clearly like that of the fork except that it does not duplicate the input. In Haskell, we define $\times$ as *cross*, below.

$$\begin{array}{lcl} cross & :: & (a \to c, b \to d) \to (a, b) \to (c, d) \\ cross\ (f, g)\ (x, y) & = & (f\, x, g\, y) \end{array}$$

We can show that $\times$ preserves identities and composition. This makes it a bifunctor. A *bifunctor* F on a category $\mathbf{C}$ is a binary operation on objects of $\mathbf{C}$ together with a binary operation on arrows of $\mathbf{C}$ such that if $f : X \to X'$ and $g : Y \to Y'$ then

$$F\ (f, g) \ : \ F\ (X, Y) \to F\ (X', Y')$$

It must also preserve identities and composition.

$$\begin{aligned} F\ (id_X, id_Y) &= id_{F\,(X,Y)} \\ F\ (f, g) \cdot F\ (h, j) &= F\ (f \cdot h, g \cdot j) \end{aligned}$$

For any bifunctor F and object A, we can obtain an endofunctor $F_{A,-}$ by fixing the first argument to be A and varying the second.

$$\begin{aligned} F_{A,-}\ (B) &= F\ (A, B) \\ F_{A,-}\ (f) &= F\ (id_A, f) \end{aligned}$$

This construction is called *left-sectioning*. Similarly, for any object B, *right-sectioning* F gives an endofunctor $F_{-,B}$ by fixing the second argument to be B and varying the first. One important bifunctor is *Outl*, defined on objects by $Outl\ (A, B) = A$ and on arrows by $Outl\ (f, g) = f$. Similarly, we define the bifunctor *Outr* by $Outr\ (A, B) = B$ and $Outr\ (f, g) = g$.

2.3.3 Coproducts

The *coproduct* of objects A and B is an object $A + B$ and two arrows, $inl_{A,B} : A \to A + B$ and $inr_{A,B} : B \to A + B$. A *join* operator must map any pair of arrows $f : A \to C$ and $g : B \to C$ to a third arrow $[f, g] : A + B \to C$, defined by the universal property

$$h = [f, g] \ \equiv \ h \cdot inl_{A,B} = f \quad \text{and} \quad h \cdot inr_{A,B} = g$$

(If the category used is $\mathbf{CPO}_\bot$ then the equation we gave for $+$ on sets does not define a true coproduct and to get its real universal property we must add to the right-hand side, the condition that h be strict.) Below, the Haskell type constructor *Either* corresponds to the operator $+$ on objects of **Fun**, and the constructor functions $Left\ :\ Either\ a\ b \to a$ and $Right\ :\ Either\ a\ b \to b$ correspond to the arrows $inl_{A,B}$ and $inr_{A,B}$ of **Fun**.

$$\mathbf{data}\ Either\ a\ b \ = \ Left\ a \mid Right\ b$$

The join operator can now be defined in Haskell as follows.

$$\begin{array}{lcl} \mathit{join} & :: & (a \to c, b \to c) \to \mathit{Either}\; a\; b \to c \\ \mathit{join}\; (f, g)\; (\mathit{Left}\; x) & = & f\; x \\ \mathit{join}\; (f, g)\; (\mathit{Right}\; x) & = & g\; x \end{array}$$

The function $\mathit{join}\; (f, g)$ replaces the constructor functions *Left* and *Right* with the functions f and g. We extend $+$ to a bifunctor with

$$f + g \;\; = \;\; [\mathit{inl} \cdot f, \mathit{inr} \cdot g]$$

Clearly, the action of $+$ on arrows is like that of the join operator except that the constructor functions are put back after they are replaced with functions. In Haskell, we define $+$ as follows:

$$\begin{array}{lcl} \mathit{sum} & :: & (a \to c, b \to d) \to \mathit{Either}\; a\; b \to \mathit{Either}\; c\; d \\ \mathit{sum}\; (f, g)\; (\mathit{Left}\; x) & = & \mathit{Left}\; (f\; x) \\ \mathit{sum}\; (f, g)\; (\mathit{Right}\; x) & = & \mathit{Right}\; (g\; x) \end{array}$$

Two useful fusion laws for coproducts are

$$k \cdot [f, g] \;\; = \;\; [k \cdot f, k \cdot g]$$

$$[f, g] \cdot (h + j) \;\; = \;\; [f \cdot h, g \cdot j]$$

Similar fusion laws exist for products but we shall not use them in this thesis. However, in Chapter 5 we shall introduce the concept of a relational product and give for it an absorption law, which is a type of fusion law.

2.3.4 Functors in general

Endofunctors and bifunctors are special cases of functors. A functor F maps objects and arrows in a source category $\mathbf{A}$ to objects and arrows in a target category $\mathbf{B}$; we write this information as $F : \mathbf{A} \to \mathbf{B}$. An endofunctor on $\mathbf{C}$ has the type $\mathbf{C} \to \mathbf{C}$. A bifunctor on $\mathbf{C}$ has the type $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$. Here, $\mathbf{C} \times \mathbf{C}$ is the product category of $\mathbf{C}$ and $\mathbf{C}$, defined below.

For any categories $\mathbf{A}$ and $\mathbf{B}$, the product category of $\mathbf{A}$ and $\mathbf{B}$ exists and is denoted $\mathbf{A} \times \mathbf{B}$. Its objects are pairs (a, b) such that a is an object of $\mathbf{A}$ and b is an object of $\mathbf{B}$. Its arrows are pairs (f, g) such that f is an arrow of

A and g is an arrow of **B**. The identity arrow is $id_{A\times B}$=(id_A, id_B) and the composition of arrows is defined by

$$(f, g) \cdot (h, k) \quad = \quad (f \cdot h, g \cdot k)$$

In general, a functor F from category **A** to category **B** is a mapping on objects combined with a mapping on arrows such that

$$\begin{aligned} F\,f \quad &: \quad F\,A \to F\,B \quad \text{if} \quad f : A \to B \\ F\;id_A \quad &= \quad id_{F\,A} \\ F\;(f \cdot g) \quad &= \quad F\,f \cdot F\,g \end{aligned}$$

We shall call the action of F on arrows the *map operator* of F and the arrows returned by F, *map operations* of F.

2.3.5 Natural transformations

The function *duplic*, below, takes a value and returns two copies of its input.

$$\begin{aligned} duplic_A \quad &: \quad A \to A \times A \\ duplic_A \quad &= \quad \langle id_A, id_A \rangle \end{aligned}$$

Note that a different function $duplic_A$ is defined for each object A. Now, *duplic* has an interesting property. If we apply a function to a value and then make two copies of it using *duplic*, we get the same result as if we had used *duplic* first and then applied the function to both copies. We have, for all $f : A \to B$,

$$(f \times f) \cdot duplic_A \quad = \quad duplic_B \cdot f$$

In fact, *duplic* is an example of a natural transformation. For any functors F and G, a *natural transformation* η *from* F *to* G is a mapping that takes each object A to an arrow $\eta_A : F\,A \to G\,A$, and that satisfies the following *naturality* property: for all arrows $f : A \to B$,

$$G\,f \cdot \eta_A \quad = \quad \eta_B \cdot F\,f$$

We write $\eta : F \dot{\to} G$ to indicate that η is a natural transformation from F to G.

Let *Id* and *Pair* be defined on objects and arrows by $Id\;X = X$ and $Pair\;X =$

$X \times X$. Then the type of *duplic* is $Id \dot{\to} Pair$. The identity arrows together form a natural transformation of type $Id \dot{\to} Id$. Our last example uses bifunctors. For all arrows $f : A \to A'$ and $g : B \to B'$, we have

$$Outl\ (f, g) \cdot outl_{A,B} \quad = \quad outl_{A',B'} \cdot (f \times g)$$

Therefore, *outl* is a natural transformation with type $(\times) \dot{\to} Outl$. These examples give us some idea what kind of functions are natural transformations. They can copy elements (like *duplic*), remove elements (like *outl*), do both or even do neither (like *id*) but they cannot create new elements or change elements. (In Chapter 7, we mention a theorem that makes this intuition more precise by relating it to the notion of membership.) Furthermore, their behaviour cannot depend on the values of elements so the function *min*, which returns the minimum of a pair, is not a natural transformation because, defining the function *neg* by $neg\, x = -x$, we can construct a counterexample.

$$(neg \cdot min)\ (3, 4) \quad \neq \quad (min \cdot (neg \times neg))\ (3, 4)$$

A natural transformation can also be subscripted by a functor. This subscripting takes it to another natural transformation as follows:

$$\eta : F \dot{\to} G \quad \Rightarrow \quad \eta_H : F \cdot H \dot{\to} G \cdot H$$

Similarly, the map operation can be lifted to natural transformations.

$$\eta : F \dot{\to} G \quad \Rightarrow \quad H\, \eta : H \cdot F \dot{\to} H \cdot G$$

2.4 Regular datatypes

2.4.1 Semantics of regular datatypes

Recall that a parameterised regular datatype T can, by definition, be put into the following canonical form.

$$T\, A \quad \approx \quad B\ (A, T\, A)$$

We shall call B the *base functor* of T. Now, B is a bifunctor so it can be left-sectioned.

$$T\, A \quad \approx \quad B_A\ (T\, A)$$

This is an instance of the more general fixpoint equation $T \approx F\ T$, where T is an object and F is an endofunctor. We shall use initial algebras to represent recursive datatypes such as T: an arrow $\alpha : F\ T \to T$ is an *initial F-algebra* if for each arrow $f : F\ Y \to Y$, there is a unique arrow $h : T \to Y$ such that

$$h \cdot \alpha \;=\; f \cdot F\, h$$

Now we can appeal to an important result.

Lambek's Lemma An arrow $\alpha : F\ T \to T$ is an isomorphism if it is an initial F-algebra.

Therefore, we can prove α is an isomorphism by showing it is an initial algebra.

2.4.2 Standard folds on lists

If we instantiate T to $\mathit{List}\ A$ and F to $\mathit{Base}_{A,-}$ then Lambek's lemma says that there is a function *foldlist* that maps arrows of type $(\mathit{Base}_{A,-})\ Y \to Y$ to unique arrows of type $\mathit{List}\ A \to Y$.

$$\begin{aligned} \mathit{foldlist} \;&:\; (\mathit{Base}\ (A, Y) \to Y) \to (\mathit{List}\ A \to Y) \\ \mathit{foldlist}\, f \cdot \alpha \;&=\; f \cdot \mathit{Base}\ (\mathit{id}, \mathit{foldlist}\, f) \end{aligned}$$

These unique arrows are called *standard folds* and *foldlist* is called a *standard fold operator*. Because *foldlist* captures a common pattern of recursion, there are many such standard folds in programs. One of them is *sumlist*, the function that adds up a list of integers. Folds are important because if we prove a theorem about *foldlist* then we also prove a fact about *sumlist* or any other standard fold.

Now we shall explain how to turn the definition of *foldlist* above into a Haskell program written in familiar pointwise style. We begin by writing both f and α as joins.

$$\begin{aligned} f = [\mathit{nil}, \mathit{cons}] \;&:\; 1 + A \times Y \to Y \\ \alpha = [\mathit{Nil}, \mathit{Cons}] \;&:\; 1 + A \times (\mathit{List}\ A) \to \mathit{List}\ A \end{aligned}$$

Now the definition of *foldlist* is

$$foldlist\,f \cdot [Nil, Cons] \;=\; [nil, cons] \cdot Base\ (id, foldlist\,f)$$

This equation can be split in two using first the fusion laws of coproducts, to give an equality of joins, and then the universal property of coproduct.

$$foldlist\ [nil, cons] : List\ A \rightarrow Y$$

$$(foldlist\ [nil, cons] \cdot Nil)\ \mathsf{1} = nil\,\mathsf{1}$$
$$(foldlist\ [nil, cons] \cdot Cons)\ (a, x)$$
$$= cons \cdot (id \times foldlist\ [nil, cons])\ (a, x)$$

Separating the join of *nil* and *cons*, writing *nil* without the $\mathsf{1}$ and using pointwise notation gives the following Haskell code.

$$\begin{array}{lcl} foldlist & :: & b \rightarrow ((a, b) \rightarrow b) \rightarrow List\ a \rightarrow b \\ foldlist\,nil\,cons\,Nil & = & nil \\ foldlist\,nil\,cons\ (Cons\ (a, x)) & = & cons\ (a, foldlist\,nil\,cons\,x) \end{array}$$

Modulo notation and currying, this is exactly the same as the familiar function *foldr*.

2.4.3 Example: summing a list

Here is the standard fold that sums a list.

$$\begin{array}{lcl} sumlist & : & List\,Int \rightarrow Int \\ sumlist & = & foldlist\ [kzero, plus] \end{array}$$

Here, *kzero* is defined by $kzero\,x = 0$ and *plus* is defined by $plus\ (x, y) = x + y$. The join of *kzero* and *plus* has the type

$$\mathsf{1} + Int \times Int \rightarrow Int$$

and that is the type required for the argument to *foldlist*. In Haskell, we write *sumlist* as follows:

$$\begin{array}{lcl} sumlist & :: & List\,Int \rightarrow Int \\ sumlist & = & foldlist\ 0\ (uncurry\ (+)) \end{array}$$

2.4.4 Example: mapping a list

We can also use *foldlist* to write the map operation for lists.

$$List\, k = foldlist\ (\alpha \cdot Base\ (k, id))$$

In Haskell, this is

$$\begin{aligned} maplist &:: (a \to b) \to (List\ a \to List\ b) \\ maplist\, f &= foldlist\ [\,]\ (\backslash(a, x) \to (f\ a, x)) \end{aligned}$$

Here, "$\backslash y \to z$" is Haskell notation for the lambda-term $\lambda y.z$.

2.4.5 Definition of regular functors

The class of *regular* endofunctors is the closure under taking least fixed points of the class of polynomial endofunctors. To motivate a definition of the latter class, consider the endofunctor $Base_A$, the least fixed point of which, *List*, is regular.

$$Base_A\, X = \mathsf{1} + A \times X$$

We will now rewrite $Base_A$ in point-free style. As we do this, we derive as steps in our calculation, the definitions for the simple functors and operators on functors that are needed to define the class of polynomial functors.

$$\begin{aligned} & Base_A\, X \\ = & \quad \{ \text{ definition of } Base_A \ \} \\ & \mathsf{1} + (A \times X) \\ = & \quad \{ \ K_1\, X = \mathsf{1}\ ;\ K_A\, X = A\ ;\ Id\, X = X \ \} \\ & (K_1\, X) + (K_A\, X \times Id\, X) \\ = & \quad \{ \ (F \times G)\, X = F\, X \times G\, X \ \} \\ & (K_1\, X) + ((K_A \times Id)\, X) \\ = & \quad \{ \ (F + G)\, X = F\, X + G\, X \ \} \\ & (K_1 + (K_A \times Id))\, X \end{aligned}$$

So we conclude

$$Base_A = K_1 + (K_A \times Id)$$

The *constant functor for* A is denoted K_A, and maps every object to A and every arrow to id_A. The *identity functor* is denoted Id, and maps every object to itself and every arrow to itself. That takes care of the simple functors from which *Base* is composed.

Now for the binary operators with which we can make functors from other simpler functors. Composition of functors is defined on objects by $(F \cdot G)\, A = F\ (G\, A)$ and on arrows by $(F \cdot G)\, f = F\ (G\, f)$. Product and coproduct were lifted to functors in the manipulation above. The following mappings for objects were used:

$$\begin{aligned}(F + G)\, A &= F\, A + G\, A \\ (F \times G)\, A &= F\, A \times G\, A\end{aligned}$$

The corresponding mappings for arrows are

$$\begin{aligned}(F + G)\, f &= F\, f + G\, f \\ (F \times G)\, f &= F\, f \times G\, f\end{aligned}$$

An endofunctor is *polynomial* if it is constructed from the identity functor and the constant functors using only the operations of coproduct, product and composition, in increasing order of precedence. Once again, the class of *regular* functors is the closure under taking least fixed points of the class of polynomial functors.

Now we shall motivate a slightly different definition of the terms polynomial and regular. The new style is closer to that needed later to define the class of nested functors. We know how to write the left-sectioning of *Base* pointwise. Now we shall write the bifunctor *Base* itself pointwise.

$$Base \;=\; KK_1 + Outl \times Outr$$

Here, KK_A is the binary variant of K_A that maps pairs of objects to A and pairs of arrows to id_A.

Since $+$ is an infix bifunctor, we shall write all expressions $F + G$ in the form $+ \cdot \langle F, G \rangle$ where

$$(H \cdot \langle F, G \rangle)\ X = H\ (F\, X, G\, X)$$

Note that F and G must have the same source categories and the same target categories. Let $\mathbf{C}^n$ abbreviate $\mathbf{C} \times \ldots \times \mathbf{C}$, where $\mathbf{C}$ appears n times. (Clearly then, $\mathbf{C}^1$=$\mathbf{C}$.) A polynomial n-ary functor on $\mathbf{C}$ is a functor of type $\mathbf{C}^n \rightarrow \mathbf{C}$ for some base category $\mathbf{C}$. It is constructed from the grammar

$$P \quad ::= \quad K_A^n \;\mid\; \Pi_i^n \;\mid\; + \;\mid\; \times \;\mid\; P_0 \cdot \langle P_1, \ldots, P_n \rangle$$

Here, n and i are natural numbers that need not be the arity of the functor. The first case generalises K and KK.

$$\begin{aligned} K_A^m \; (A_1, \ldots, A_m) &= A \\ K_A^m \; (f_1, \ldots, f_m) &= id_A \end{aligned}$$

The second case generalises *Id* and *Outl* and *Outr* to give a *projection functor*, Π_i^m.

$$\begin{aligned} \Pi_i^m \; (A_1, \ldots, A_i, \ldots, A_m) &= A_i \\ \Pi_i^m \; (f_1, \ldots, f_i, \ldots, f_m) &= f_i \end{aligned}$$

Finally, the last case generalises composition to an n-ary functor F,

$$(F \cdot \langle G_1, \ldots, G_n \rangle) \, X \quad = \quad F \; (G_1 \, X, \ldots, G_n \, X)$$

This grammar is versatile enough to deal with datatypes that have several parameters, as we shall verify in the next section, when we use a similar grammar to define the class of nested functors.

2.4.6 Generic standard fold operator

Suppose that $T\,A$ is a regular parameterised datatype and the least fixed point of $B_{A,-}$, for some bifunctor B. Suppose further that the initial algebra of B_A is $\alpha : B\;(A, T\,A) \rightarrow T\,A$. Then the standard fold operator for T is given by $fold_{B_{A,-}}$, which is defined by the universal property

$$\begin{aligned} fold_{B_{A,-}} \quad &: \quad (B\;(A, Y) \rightarrow Y) \rightarrow T\,A \rightarrow Y \\ h = fold_{B_{A,-}}\, f \quad &\equiv \quad h \cdot \alpha = f \cdot B\;(id, h) \end{aligned}$$

If we work in the category $\mathbf{CPO}_\perp$ then we must add to the right-hand side the extra condition that f and h should both be strict.

2.5 Nested datatypes

2.5.1 Semantics of nested datatypes

Our running example shall be the datatype of nests defined by

$$\textbf{data}\ \mathit{Nest}\ a \quad = \quad \mathit{NilN} \mid \mathit{ConsN}\ (a, \mathit{Nest}\ (\mathit{Pair}\ a))$$

A nest is a list of power trees where each successive power tree has height one greater than its predecessor. It can be written in category notation as follows, using the bifunctor *Base* that was used for lists.

$$\mathit{Nest}\ A \quad \approx \quad \mathit{Base}\ (A, \mathit{Nest}\ (\mathit{Pair}\ A))$$

However, Lambek's lemma only asserts isomorphisms of the form $T \approx F\ T$. Here, quite inconveniently, it is the functor *Nest* that appears on both sides, not the object $\mathit{Nest}\ A$. Therefore, we need a notion of an isomorphism between functors, also denoted $\approx$ and defined later, so that we can define *Nest* as a fixpoint instead. This is the approach is taken in [BM98] and [BP99b]. For an appropriate mapping *NestF* from endofunctors to endofunctors, we have

$$\mathit{Nest} \quad \approx \quad \mathit{NestF}\ \mathit{Nest}$$

Now we must find *NestF*. Putting the two isomorphisms together suggests

$$\mathit{NestF}\ \mathit{Nest}\ A \quad = \quad \mathit{Base}\ (A, \mathit{Nest}\ (\mathit{Pair}\ A))$$

Written point-free, this is

$$(\mathit{NestF}\ \mathit{Nest})\ A \quad = \quad (\mathit{Base} \cdot \langle \mathit{Id}, \mathit{Nest} \cdot \mathit{Pair} \rangle)\ A$$

Abstracting the above suggests

$$\mathit{NestF}\ X \quad = \quad \mathit{Base} \cdot \langle \mathit{Id}, X \cdot \mathit{Pair} \rangle$$

2.5.2 The endofunctor category

To prove the isomorphism between functors, we show that it is an initial algebra. The category in which it is an arrow is **Nat** (**Fun**). Here, **Nat** (**C**) denotes the *endofunctor category* formed from **C**. The objects of this category are endofunctors on **C** and the arrows are natural transformations between these endofunctors. The identity arrow for an object F is the natural

transformation id_F defined by $(id_F)_A = id_{F\,A}$. The composition of two natural transformations $\alpha : G \dot{\to} H$ and $\beta : F \dot{\to} G$ is $\alpha{\cdot}\beta$ where $(\alpha{\cdot}\beta)_A = \alpha_A{\cdot}\beta_A$.

Isomorphism between endofunctors on $\mathbf{C}$ is now simply given by the categorical notion of isomorphism between the objects of $\mathbf{Nat}\ (\mathbf{C})$. Endofunctors on $\mathbf{Nat}\ (\mathbf{C})$ such as *NestF* are called higher-order functors, or *hofunctors* for short.

Product and fork are defined in $\mathbf{Nat}\ (\mathbf{C})$ by lifting from $\mathbf{C}$.

$$
\begin{aligned}
(F \times G)\, X &= (F\, X) \times (G\, X) \\
(outl_{F,G})_A &= outl_{F\,A, G\,A} \\
(outr_{F,G})_A &= outr_{F\,A, G\,A} \\
(\langle \eta, \theta \rangle)_A &= \langle \eta_A, \theta_A \rangle
\end{aligned}
$$

The action of product on arrows is defined by

$$(\eta \times \theta)_A = \eta_A \times \theta_A$$

Coproduct and join are defined similarly.

2.5.3 Simple folds for nests

Lambek's lemma informs us that $\alpha : \mathit{NestF}\ \mathit{Nest} \dot{\to} \mathit{Nest}$ is an isomorphism as required if it is an initial algebra in $\mathbf{Nat}\ (\mathbf{Fun})$. We shall confirm that α is an initial algebra but first let us look at what this means. A function *hfoldnest* must map each arrow in $\mathbf{Nat}\ (\mathbf{Fun})$ with type $\mathit{NestF}\ R \dot{\to} R$ to a unique arrow of type $\mathit{Nest} \dot{\to} R$.

$$
\begin{aligned}
\mathit{hfoldnest} &: (\mathit{NestF}\ R \dot{\to} R) \to (\mathit{Nest} \dot{\to} R) \\
\mathit{hfoldnest}\, f \cdot \alpha &= f \cdot \mathit{NestF}\ (\mathit{hfoldnest}\, f)
\end{aligned}
$$

We call this function the *simple fold operator* for nests. (The prefix 'h' is there to remind us that the fold is constructed in a higher-order category.) The type of the argument f can be written using pointwise notation as follows

$$f_A \ :\ \mathsf{1} + A \times\ R\ (\mathit{Pair}\ A) \to R\ A$$

Turning *hfoldnest* into Haskell is easy, given its similarity to *foldlist*. However, we need a special typing syntax for both the functor variable R and the

universal quantifier, which has a local scope. These two features are allowed in Haskell '98 as part of what is known as a *rank two* type signature [JL]:

$$\begin{aligned} \mathit{hfoldnest} \; :: \; & (\textbf{forall}\, b.\, r\ b) \rightarrow \\ & (\textbf{forall}\, b.\, (b, r\ (\mathit{Pair}\ b)) \rightarrow r\ b) \rightarrow \\ & \mathit{Nest}\ a \rightarrow r\ a \end{aligned}$$

$$\begin{aligned} \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ \mathit{NilN} &= \mathit{nil} \\ \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ (\mathit{ConsN}\ (a, x)) &= \mathit{cons}\ (a,\ \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ x) \end{aligned}$$

The definition of *hfoldnest* contains polymorphic recursion, so we must attach a type signature. As the arguments are natural transformations, being arrows of **Nat** (**Fun**), the type signature must be rank two. Helpfully, we can reach this conclusion without thinking categorically. Suppose a fold replaces the constructors *NilN* and *ConsN* with themselves. Then the function *nil* can be used with the type $\mathit{Nest}\ a$ or $\mathit{Nest}\ (\mathit{Pair}\ a)$ and so on, depending on how deeply nested is the *NilN* to be replaced. Similarly, when *hfoldnest* is used to flatten a nest, *nil* can be used with type $[a]$ or $[\mathit{Pair}\ a]$ and so on. Giving *nil* the type $\textbf{forall}\, b.\, r\ b$, using both type constructors and local universal quantifiers, covers all these possibilities.

It must be noted that the variables a and r in the type signature of *hfoldnest* are also universally quantified. We shall in future make this explicit, for the sake of clarity, by adding **forall**'s at the outermost with global scope; the latest versions of Haskell allow us to do this.

$$\begin{aligned} \mathit{hfoldnest} \; :: \; & \textbf{forall}\, a\, r. \\ & (\textbf{forall}\, b.\, r\ b) \rightarrow \\ & (\textbf{forall}\, b.\, (b, r\ (\mathit{Pair}\ b)) \rightarrow r\ b) \rightarrow \\ & \mathit{Nest}\ a \rightarrow r\ a \end{aligned}$$

$$\begin{aligned} \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ \mathit{NilN} &= \mathit{nil} \\ \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ (\mathit{ConsN}\ (a, x)) &= \mathit{cons}\ (a,\ \mathit{hfoldnest}\ \mathit{nil}\ \mathit{cons}\ x) \end{aligned}$$

2.5.4 Blampied's algebra family folds

Paul Blampied observed in his thesis [Bla00] that the only instances of the polymorphic function *nil* that are actually needed by *hfoldnest* are those with types belonging to the family $r\ (\mathit{Pair}^k\ a)$, for some a. This observation motivates his algebra family folds, which take as a parameter an infinite data

structure, called an algebra family, containing every possible value for the constructor functions. This structure, effectively an infinite list, is traversed as the nest is traversed so that the correct version of *nil* and *cons* is used at each stage.

The folds are constructed in a category whose objects are functions from I to objects of **Fun**, or any other suitable category, and whose arrows are functions from I to arrows of **Fun**. Here, I is an indexing set for all the possible replacements for each function. For nests, I is the set $\{Id, Pair, Pair{\cdot}Pair, \ldots\}$.

We shall see later that the natural transformation arguments we give to *nil* and *cons* have types too limited for certain operations such as summation. This leads us to introduce more complicated generalised and efficient folds in the next chapter. This trouble can be avoided if we stick with Blampied's folds but they have severe drawbacks. Although the definition of the fold operator itself is simple, the algebra families themselves are difficult to construct, and in the case of higher-order nested datatypes, Blampied hints that rank three polymorphism is required.

Consequently, the fusion laws depend upon an infinite family of conditions. Although it is often possible to prove that these conditions hold, as when Blampied shows the map operators he defines for nested datatypes are functors, such proofs require us to use extra information about the particular algebra family in an ad hoc way. Blampied only shows how this can be done for one example. Since fusion laws are absolutely integral to this thesis, we shall stick with simple and generalised folds and use the fusion laws given by Bird and Paterson in [BP99b].

2.5.5 Example: Flattening a nest

To *flatten* a data structure is to turn it into a list of its elements with the same syntactic order. The function that flattens a nest can be written using *hfoldnest*.

$$\begin{array}{lcl} hflatnest & :: & Nest\ a \rightarrow [a] \\ hflatnest & = & hfoldnest\ [\,]\ fcons \end{array}$$

The empty nest *NilN* is replaced by the empty list. (We prefer the notation of Haskell's built-in lists to that of our own datatype *List*). The *ConsN*

constructors are replaced by *fcons* where

$$\begin{aligned} fcons &:: (b, [Pair\ b]) \to [b] \\ fcons\ (b, ys) &= b : flatlp\ ys \end{aligned}$$

To flatten a list of pairs, we flatten every pair to give a list of two-element lists, which we can then concatenate.

$$\begin{aligned} flatlp &:: [Pair\ a] \to [a] \\ flatlp &= concat \cdot map\ (\backslash\, (x, y) \to [x, y]) \end{aligned}$$

Here is a sample evaluation for *hflatnest*; nests are written with a list-like notation for conciseness.

$$\begin{aligned} & hflatnest \ll 1, (2,3), ((4,5),(6,7)) \gg \\ = \;& 1 : flatlp\ ((2,3) : flatlp\ (((4,5),(6,7)) : flatlp \ll\gg)) \\ = \;& 1 : flatlp\ ((2,3) : flatlp\ (((4,5),(6,7)) : [\,])) \\ = \;& 1 : flatlp\ ((2,3) : flatlp\ [((4,5),(6,7))]) \\ = \;& 1 : flatlp\ [(2,3),(4,5),(6,7)] \\ = \;& [1,2,3,4,5,6,7] \end{aligned}$$

The function *hflatnest* comes from [BM98]. Another way to flatten a nest will be derived in Chapter 3.

2.5.6 Existence of initial algebras

Now we want to call upon some result that says that all polynomial hofunctors in **Nat** (**Fun**) have initial algebras. Gibbons and Martin [MG01] try to prove a sufficient condition: that all polynomial hofunctors are ω-cocontinuous. (See [MG01] for the definition of the term ω-cocontinuous.) They build on Blampied's proof for the special case of linear datatypes [Bla00]. Their proof is by structural induction but one of their inductive cases only preserves ω-cocontinuity for hofunctors in the smaller category **Coc** (**Fun**) contained within **Nat** (**Fun**). Here, **Coc** (**C**) is the category of ω-cocontinuous endofunctors on **C**.

The proof in [MG01] is for any category **Coc** (**C**) where **C** is any ω-cocomplete (defined in [MG01]) category that has products and coproducts for any pair of objects and also an initial object. An *initial object* is an object that has

exactly one arrow from it to every object. The category **Fun** has the empty set as an initial object and it satisfies the other conditions so all polynomial hofunctors in **Coc** (**Fun**) have initial algebras.

2.5.7 Definition of nested functors

A *nested* functor is the least fixed point of a polynomial hofunctor. An example of a polynomial hofunctor is

$$\begin{aligned} \mathit{NestF}\, X &= \mathit{Base} \cdot \langle \mathit{Id}, X \cdot \mathit{Pair} \rangle \\ \mathit{Base} &= KK_1 + \mathit{Outl} \times \mathit{Outr} \\ \mathit{Pair} &= \mathit{Id} \times \mathit{Id} \end{aligned}$$

A hofunctor F is *polynomial* if its definition has the form $F\, X = P$ where P is a term of the following grammar

$$P \quad ::= \quad K^n_A \;\mid\; \Pi^n_i \;\mid\; + \;\mid\; \times \;\mid\; P_0 \cdot \langle P_1, \ldots, P_m \rangle \;\mid\; X$$

Observe that we have simply augmented the previous grammar with the variable X as a sixth case so that the parameter of the hofunctor can appear on the right-hand side. However, since regular datatypes are defined as the *closure* of polynomial functors under least fixed points, it follows that not all regular functors are nested. Of course, because of the sixth case, some nested functors are not regular and we call these *properly nested* functors. Note that to define a generic operation for all nested datatypes, we must supply separate polynomial cases and nested fixpoint cases.

Our definition of the class of nested functors is the same as in [BP99b]. However, Gibbons and Martin give a definition of the class of polynomial hofunctors (from which nested functors are built) that looks quite different from ours but which is effectively the same. An example will make clear both the difference and the effective similarity. The polynomial hofunctor *NestF* is defined by

$$\mathit{NestF} \quad = \quad K_{K_1} + K_{\mathit{Id}} \times \mathit{Id} \star K_{\mathit{Pair}}$$

Here, $+$ and $\times$ are the coproduct and product bifunctors lifted twice (to functors and thence to hofunctors). Similarly, K and Id (at the top-level) denote the constant functor and identity functor lifted twice to hofunctors.

Finally, $\star$ denotes the horizontal composition bifunctor lifted once to hofunctors. The unlifted version is defined as follows. For endofunctors P and Q we have,

$$P \star Q \quad = \quad P \cdot Q$$

For natural transformations $\theta : P \dot{\to} R$ and $\psi : Q \dot{\to} S$, we have

$$\theta \star \psi \;=\; \theta_S \cdot P\psi \;=\; R\psi \cdot \theta_Q$$

Now we demonstrate that the definition we just gave for *NestF* matches the version we have been using in this chapter.

$$\begin{array}{ll}
 & \mathit{NestF}\ X \\
= & (K_{K_1} + K_{Id} \times Id \star K_{Pair})\ X \\
= & K_{K_1}\ X + K_{Id}\ X \times (Id \star K_{Pair})\ X \\
= & K_1 + Id \times (Id\ X) \cdot (K_{Pair}\ X) \\
= & K_1 + Id \times X \cdot Pair \\
= & Base \cdot \langle Id, X \cdot Pair \rangle
\end{array}$$

Gibbons and Martin define the class of polynomial hofunctors as being constructed from the identity and constant hofunctors, composition, lifted horizontal composition, and the twice-lifted product and coproduct bifunctors. Any polynomial (according to our definition) hofunctor can be put in this form so results about the existence of initial algebras in [MG01] and [BGM] can be transported to this thesis. However, to convert between the two different forms, we must first rewrite the functor equations so as to eliminate all uses of the fork operator at the top-level.

2.5.8 Simple folds on alternating lists

Now consider the datatype of alternating lists, which has more than one parameter.

$$\textbf{data}\ AL\ a\ b \quad = \quad NilAL \mid ConsAL\ (a, AL\ b\ a)$$

We can write this in category notation as follows.

$$AL\ (A, B) \quad \approx \quad 1 + A \times AL\ (Swap\ (A, B))$$

The functor *Swap* is defined by $Swap\ (A, B) = (B, A)$. Note that AL is not an endofunctor but a bifunctor. Therefore, we shall construct it as an object in

the category of bifunctors on **Fun**. (In general, a category of n-ary functors on **Fun** is needed for a datatype that has n parameters). We can start by lifting the isomorphism between objects to an isomorphism between functors:

$$AL \approx KK_1 + \mathit{Outl} \times AL \cdot \mathit{Swap}$$

Now we rewrite AL as follows:

$$\begin{array}{rcl} AL & \approx & ALF\,AL \\ ALF\,X & = & KK_1 + \mathit{Outl} \times X \cdot \mathit{Swap} \end{array}$$

The parameter to the simple fold operator is a natural transformation f with components $f_{A,B}$ of type

$$ALF\ R\ (A, B) \to R\ (A, B)$$

According to the definition of ALF, this type is equal to

$$1 + A \times R\ (B, A) \to R\ (A, B)$$

This information enables us to write the fold operator in Haskell.

$$\begin{array}{lll} \mathit{hfoldal} & :: & \mathbf{forall}\ c\ d\ r. \\ & & (\mathbf{forall}\ a\ b.\ r\ a\ b) \to \\ & & (\mathbf{forall}\ a\ b.\ (a, r\ b\ a) \to r\ a\ b) \to \\ & & AL\ c\ d \to r\ c\ d \end{array}$$

$$\begin{array}{lcl} \mathit{hfoldal\ nil\ cons\ NilAL} & = & \mathit{nil} \\ \mathit{hfoldal\ nil\ cons}\ (\mathit{ConsAL}\,(a,\ x)) & = & \mathit{cons}\ (a,\ \mathit{hfoldal\ nil\ cons}\ x) \end{array}$$

Before giving an example use of *hfoldal*, we shall first demonstrate that ALF can be written using the grammar for polynomial hofunctors.

$$\begin{array}{rcl} ALF\,X & = & \mathit{BaseAL} \cdot \langle \mathit{Id}, X \cdot \mathit{Swap} \rangle \\ \mathit{BaseAL} & = & KK_1 + \mathit{Outl} \cdot \mathit{Outl} \times \mathit{Outr} \\ \mathit{Swap} & = & \langle \mathit{Outr}, \mathit{Outl} \rangle \end{array}$$

To check that AL is the least fixed point of this hofunctor ALF, we apply it to a pair of objects.

$$\begin{array}{ll} & AL\ (A, B) \\ \approx & ALF\ AL\ (A, B) \end{array}$$

$$
\begin{aligned}
&= \mathit{BaseAL} \cdot \langle \mathit{Id}, \mathit{AL} \cdot \langle \mathit{Outr}, \mathit{Outl} \rangle \rangle \ (A, B) \\
&= \mathit{BaseAL} \ (\mathit{Id} \ (A, B), \mathit{AL} \ (\langle \mathit{Outr}, \mathit{Outl} \rangle \ (A, B)) \\
&= \mathit{BaseAL} \ ((A, B), \mathit{AL} \ (B, A)) \\
&= (\mathit{KK}_1 + \mathit{Outl} \cdot \mathit{Outl} \times \mathit{Outr}) \ ((A, B), \mathit{AL} \ (B, A)) \\
&= 1 + A \times \mathit{AL} \ (B, A)
\end{aligned}
$$

2.5.9 Example: separating an alternating list

We can use a simple fold on lists to separate an alternating list into a pair of lists having different types.

$$
\begin{aligned}
&\mathit{separate} &&:: \ \mathit{AL}\, a\, b \to ([a], [b]) \\
&\mathit{separate} &&= \ \mathit{hfoldal} \ ([\,], [\,]) \ (\backslash (a, (x, y)) \to (a : y, x))
\end{aligned}
$$

Unfortunately, as it stands, this program produces a type error. This is because Haskell compilers and interpreters use first-order matching and unification when comparing type signatures. This means that functor variables like r can only be bound to (possibly curried) named type constructors like [–] and (–,–). So a functor variable can be bound to $\lambda x. [x]$ or $\lambda x, y. (x, y)$, for example, but not to $\lambda x, y. [(x, y)]$ as is required by the function *separate*. A simple solution to this problem is to specialise *hfoldal* to make r be a composition.

$$
\begin{aligned}
\mathit{hfoldal}' \ :: \ & \mathbf{forall}\ c\ d\ r\ s\ t. \\
& (\mathbf{forall}\ a\ b.\ r\ (s\ a)\ (t\ b)) \to \\
& (\mathbf{forall}\ a\ b. (a, r\ (s\ b)(t\ a)) \to r\ (s\ a)(t\ b)) \to \\
& \mathit{AL}\, c\, d \to r\ (s\ c)\ (t\ d)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hfoldal}'\ \mathit{nil}\ \mathit{cons}\ \mathit{NilAL} &&= \ \mathit{nil} \\
&\mathit{hfoldal}'\ \mathit{nil}\ \mathit{cons}\ (\mathit{ConsAL}\ (a, x)) &&= \ \mathit{cons}\ (a, \mathit{hfoldal}'\ \mathit{nil}\ \mathit{cons}\ x)
\end{aligned}
$$

Now we can separate an alternating list using the specialised version of *hfoldal*.

$$
\begin{aligned}
&\mathit{separate}' &&:: \ \mathit{AL}\, a\, b \to ([a], [b]) \\
&\mathit{separate}' &&= \ \mathit{hfoldal}' \ ([\,], [\,]) \ (\backslash (a, (x, y)) \to (a : y, x))
\end{aligned}
$$

However, in the next chapter, we shall discover how to use *hfoldal* without rewriting it.

2.5.10 Generic simple fold operator

Suppose that $\alpha : F\,T \dot{\rightarrow} T$ is the initial algebra of a polynomial hofunctor F with least fixed point T. Then the simple fold operator for T, denoted hfold_F, is defined by the universal property

$$\begin{array}{rcl} \mathit{hfold}_F & : & (F\,R \dot{\rightarrow} R) \rightarrow T \dot{\rightarrow} R \\ h = \mathit{hfold}_F\,f & \equiv & h \cdot \alpha \;=\; f \cdot F\,h \end{array}$$

Strictly speaking, α should also be subscripted by F since it is dependent on F but we shall omit the subscript when it is obvious from context. An exception must be made to this rule, however, when the initial algebra is subscripted with a second functor to give a new natural transformation. Then we shall avoid ambiguity by including both subscripts. For example,

$$(\alpha_{\mathit{NestF}})_{\mathit{Pair}} \;:\; (\mathit{NestF}\ \mathit{Nest}) \cdot \mathit{Pair} \rightarrow \mathit{Nest} \cdot \mathit{Pair}$$

For regular datatypes, the simple fold and standard fold operators have similar definitions. Only the category used and consequent typing distinguishes the two.

For the definition above to make sense, we must define the notion of equality for natural transformations. Two natural transformations are equal exactly when their components are equal.

$$\eta = \theta \;\equiv\; \forall A : \eta_A = \theta_A$$

Indeed all the equalities in our thesis shall be between natural transformations. Since operators like product are also defined by lifting, the equations that follow are unchanged when both sides are applied to objects to give arrows of **Fun**, and in fact they are best interpreted this way.

2.5.11 Simple folds for square matrices

We close this section by considering higher-order datatypes like that of square matrices below. The way that *Square* gives exactly all the square matrices was explained in Chapter 1.

$$\begin{array}{lcl} \textbf{type}\ \mathit{Square}\ a & = & \mathit{SM}\ \mathit{Nil}\ a \\ \textbf{data}\ \mathit{SM}\ f\ a & = & \mathit{ZeroSM}\ \ (f\ \ (f\ a)) \\ & | & \mathit{SuccSM}\ \ (\mathit{SM}\ \ (\mathit{Cons}\,f)\ a) \end{array}$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Nil}\ a & = & \mathit{Nil} \\
\textbf{data}\ \mathit{Cons}\ f\ a & = & \mathit{Cons}\ a\ (f\ a)
\end{array}
$$

Since SM takes a type constructor as a parameter, it must correspond to a hofunctor and be an object of **Coc** (**Coc** (**Fun**)). We want there to be an endofunctor SMF on this category such that the arrow $\alpha : \mathit{SMF}\ \mathit{SM} \rightarrow \mathit{SM}$ is an initial algebra. According to [MG01], higher-order polynomial hofunctors (like SMF) do indeed have initial algebras. A simple fold operator for square matrices immediately follows from the definition of initial algebras, and is written in Haskell as follows.

$$
\begin{array}{lcl}
\mathit{hfoldsm} & :: & \textbf{forall}\ g\ a. \\
 & & (\textbf{forall} f\ b. f\ (f\ b) \rightarrow y\, f\ b) \rightarrow \\
 & & (\textbf{forall} f\ b. y\ (\mathit{Cons}\, f)\ b \rightarrow y\, f\ b) \rightarrow \\
 & & \mathit{SM}\ g\ a \rightarrow\ y\ g\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{hfoldsm}\ \mathit{zero}\ \mathit{succ}\ (\mathit{Zero}\ x) & = & \mathit{zero}\ x \\
\mathit{hfoldsm}\ \mathit{zero}\ \mathit{succ}\ (\mathit{Succ}\ y) & = & \mathit{succ}\ (\mathit{hfoldsm}\ \mathit{zero}\ \mathit{succ}\ y)
\end{array}
$$

If we want to use *hfoldsm* to flatten a square matrix, then y must ignore its arguments, so we need $y\ g\ a = [a]$. However, the *succ* parameter would then have the type of an identity function so we cannot use *hfoldsm* to flatten a square matrix. Unfortunately, it is hard to think how y could be anything other than a constant functor for any function we might want to write as a simple fold and since the *succ* parameter must again have the type of an identity function, it appears that *hfoldsm* has no applications.

Higher-order nested datatypes are also discussed in the conclusion of Blampied's thesis [Bla00]. Although the idea of algebra family folds can be extended to higher-order nested datatypes, it appears that the algebra families themselves will have rank three polymorphism, which is not a feature of the Haskell language.

Chapter 3

Other folds for linear datatypes

In this chapter, we shall explain why simple folds are not general enough and motivate a generalised fold operator for nests. We shall also derive an (even more general) efficient fold operator for nests. The two operators can be generalised to all linear datatypes and for some of these, efficient folds are more efficient than generalised folds. The next chapter generalises the operators to non-linear datatypes and the chapter after that extends the operators to relations. After that, we can apply our theory to some example generic operations.

Recall that we used a standard fold to sum a list of integers:

$$\begin{array}{lcl} \mathit{sumlist} & : & \mathit{List}\,\mathit{Int} \to \mathit{Int} \\ \mathit{sumlist} & = & \mathit{fold}_{\mathit{Base}_{\mathit{Int}}}\,[\mathit{kzero}\,,\,\mathit{plus}] \end{array}$$

Recall too that kzero is defined by $\mathit{kzero}\,x = 0$ and that plus is defined by $\mathit{plus}\ (x, y) = x + y$. Unfortunately, we cannot use the simple fold operator to sum a nest of integers because it has too specialised a type:

$$\begin{array}{lcl} \mathit{hfold}_{\mathit{NestF}} & : & (\mathit{Base} \cdot \langle \mathit{Id}, R \cdot \mathit{Pair}\rangle \stackrel{.}{\to} R) \to (\mathit{Nest} \stackrel{.}{\to} R) \\ \mathit{hfold}_{\mathit{NestF}}\ f \cdot \alpha & = & f \cdot \mathit{Base}\ (\mathit{id},\, \mathit{hfold}_{\mathit{NestF}}\ f) \end{array}$$

We could take R to be K_{Int}, the constant functor that maps every type to Int, but then f_A would have the type $\mathit{Base}\ (A, \mathit{Int}) \to \mathit{Int}$. Any function of this type, when given a pair, must behave independently of the type of the pair's left component. So it cannot inspect both components of its input and it cannot be equal to $[\mathit{kzero}, \mathit{plus}]$. We must therefore adapt $\mathit{hfold}_{\mathit{NestF}}$

to give a more general operator $gfold_{NestF}$ that returns generalised folds.

The structure of this chapter is as follows. Section 3.1 defines the generalised fold operator for nests. Section 3.2 derives an efficient fold operator for nests. Section 3.3 derives a fold-equality law that gives conditions when a simple fold is equal to a special case of efficient folds known as an efficient reduction. Although we can always use efficient folds in place of generalised folds, we nevertheless spend a whole section on generalised folds because the fusion laws for generalised folds are needed to prove the fusion laws for efficient folds.

Section 3.4 generalises the previous three sections from nests to linear nested datatypes and briefly discusses multi-parameter and higher-order datatypes. For simplicity, we shall ignore such datatypes for the rest of the thesis. Section 3.5 proves that generalised and efficient folds satisfy the universal properties given for them in Sections 3.1 and 3.2. Finally, Section 3.6 gives a simple map-fusion law for efficient folds.

3.1 Generalised folds for nests

3.1.1 Generalised fold operator for nests

First, we shall replace Id in the type of $hfold_{NestF}$'s parameter by a variable M to give

$$Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R$$

We can bind both M and R to K_{Int}, to give the type $Base\ (Int, Int) \to Int$. So now we can supply $[kzero, plus]$ as a parameter to the fold operator. Because we have broadened the type of the input, we must also broaden the type of the output.

$$gfold_{NestF} \ : \ (Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R) \to (Nest \cdot M \dot{\to} R)$$

The equation for $gfold_{NestF}$ cannot simply be a copy of the equation for $hfold_{NestF}$ because the two sides would then have different types:

$$\begin{array}{lcl} gfold_{NestF}\ f \cdot (\alpha_{NestF})_M & : & Base \cdot \langle M, Nest \cdot Pair \cdot M \rangle \dot{\to} R \\ f \cdot Base\ (id,\ gfold_{NestF}\ f) & : & Base \cdot \langle M, Nest \cdot M \cdot Pair \rangle \dot{\to} R \end{array}$$

Therefore we must, before recursion, perform a map operation on nests with a function g that should be supplied as an auxiliary parameter.

$$g : Pair \cdot M \dot{\to} M \cdot Pair$$

For concision, we shall write $(\!|f \mid g|\!)_{NestF}$ in place of $gfold_{NestF}\, f\, g$ and $(\!|f|\!)_{NestF}$ in place of $hfold_{NestF}\, f$.

$$\begin{array}{lcl} (\!|-|-|\!)_{NestF} & : & (Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R) \to \\ & & (Pair \cdot M \dot{\to} M \cdot Pair) \to \\ & & (Nest \cdot M \dot{\to} R) \end{array}$$

$$h = (\!|f \mid g|\!)_{NestF} \equiv h \cdot (\alpha_{NestF})_M = f \cdot Base\ (id,\ h \cdot Nest\, g)$$

(In future, we shall omit the subscript M despite its important role in making the typing clearer.) Note that generalised folds satisfy a universal property just as simple folds do. A proof of this crucial fact was provided by Bird and Paterson in [BP99b]; we shall sketch the proof at the end of this chapter.

We can recover the simple fold operator by taking $M = Id$ and $g = id_{Pair}$, so the generalised fold operator is indeed a generalisation.

$$\begin{array}{lcl} (\!|f|\!)_{NestF} & : & (NestF\ R \dot{\to} R) \to (Nest \dot{\to} R) \\ (\!|f|\!)_{NestF} & = & (\!|f \mid id|\!)_{NestF} \end{array}$$

3.1.2 Generalised fold operators in Haskell

To implement this generalised fold operator in Haskell, we must first implement the map operator for nests. We cannot use a simple fold operator for this because no simple fold can have the type $Nest \cdot K_a \dot{\to} Nest \cdot K_b$. Generalised folds are general enough but we do not yet have an operator with which to produce them, so we must define the map operator *nest* by explicit recursion:

$$\begin{array}{lcl} pair & :: & (a \to b) \to Pair\ a \to Pair\ b \\ pair\, f\ (x, y) & = & (f\ x, f\ y) \end{array}$$

$$\begin{array}{lcl} nest & :: & (a \to b) \to Nest\ a \to Nest\ b \\ nest\ f\ NilN & = & NilN \\ nest\ f\ (ConsN\ (a, x)) & = & ConsN\ (f\ a, nest\ (pair\, f)\ x) \end{array}$$

The definition of *nest* follows from the naturality property of α, which says that for all $h : a \to b$ we have

$$Nest\,h \cdot (\alpha_{NestF})_a \quad = \quad (\alpha_{NestF})_b \cdot Base\ (h,\ Nest\ (Pair\,h))$$

Now we can use *nest* to implement the generalised fold operator on nests:

$$\begin{array}{lll} gfoldnest & :: & \textbf{forall}\,m\,r\,b. \\ & & (\textbf{forall}\,a.\,r\ a) \to \\ & & (\textbf{forall}\,a.\,(m\ a, r\ (Pair\ a)) \to r\ a) \to \\ & & (\textbf{forall}\,a.\,Pair\ (m\ a) \to m\ (Pair\ a)) \to \\ & & Nest\ (m\ b) \to r\ b \end{array}$$

$$\begin{array}{l} gfoldnest\,nil\,cons\,bin\,NilN\ =\ nil \\ gfoldnest\,nil\,cons\,bin\ (ConsN\ (a, x)) \\ \qquad =\ cons\ (a,\ gfoldnest\,nil\,cons\,bin\ (nest\,bin\,x)) \end{array}$$

Examining the Haskell code, we see that generalised folds replace constructors with functions, just as simple folds do, but they also rearrange or simplify what remains of the input before each recursive subcall. To define *hfoldnest* using *gfoldnest*, we must match $Nest\,(m\,b)$ with $Nest\,b$. Since higher-order matching is needed to find the necessary binding of $m = \lambda x.x$, we could specialise the type signature of *gfoldnest* to a new function $gfoldnest'$.

$$\begin{array}{lll} gfoldnest' & :: & \textbf{forall}\,b\,r. \\ & & (\textbf{forall}\,a.\,r\ a) \to \\ & & (\textbf{forall}\,a.\,(a, r\ (Pair\ a)) \to r\ a) \to \\ & & (\textbf{forall}\,a.\,Pair\ a \to Pair\ a) \to \\ & & Nest\ b \to r\ b \end{array}$$

$$\begin{array}{l} gfoldnest\,nil\,cons\,bin\,NilN\ =\ nil \\ gfoldnest\,nil\,cons\,bin\ (ConsN\ (a, x)) \\ \qquad =\ cons\ (a,\ gfoldnest\,nil\,cons\,bin\ (nest\,bin\,x)) \end{array}$$

Now we can define *hfoldnest* by

$$hfoldnest\,nil\,cons \quad = \quad gfoldnest\,nil\,cons\,id$$

However, it is a lot of trouble to rewrite the type of a supposedly general purpose function like *gfoldnest* every time we want to use it, so we shall instead build a type constructor that mimics the functor *Id*.

$$\textbf{newtype}\ Id\ a = Id\ a$$

The **newtype** declaration asserts that $Id\ a$ is isomorphic to a. Now first-order matching will bind m to Id in the program below.

$$\begin{array}{lll} hfoldnest & :: & \mathbf{forall}\ b\ r. \\ & & (\mathbf{forall}\ a.\ r\ a) \rightarrow \\ & & (\mathbf{forall}\ a.\ (a, r\ (Pair\ a)) \rightarrow r\ a) \rightarrow \\ & & Nest\ b \rightarrow r\ b \end{array}$$

$$\begin{array}{l} hfoldnest\ nil\ cons = gfoldnest\ nil\ cons'\ bin \cdot nest\ Id \\ \quad \mathbf{where}\ cons'\ (Id\ x,\ y) = cons\ (x,\ y) \end{array}$$

It should be noted that the use of the Id constructor functions need not add to the running time of *hfoldnest* because the compiler can easily remove all occurrences of Id before executing the program.

3.1.3 Example: summing a nest

As promised at the start of this chapter, we can now use the generalised fold operator to sum a nest of integers. Generalised folds have types of the form $T \cdot M \dot{\rightarrow} R$ and by taking $M = R = K_{Int}$, we get a type $T\ Int \rightarrow Int$.

$$\begin{array}{lll} sumnest & : & Nest\ Int \rightarrow Int \\ sumnest & = & (\!|kzero, plus \mid plus|\!)_{NestF} \end{array}$$

The first argument is the join of two functions $[kzero, plus]$ with the square brackets left implicit for the sake of neatness. Below is a sample evaluation of *sumnest* to show how generalised folds work.

$$\begin{array}{ll} & sumnest\ \ll 1, (2,3), ((4,5),(6,7)) \gg \\ = & 1 + sumnest\ (nest\ plus \ll (2,3), ((4,5),(6,7)) \gg) \\ = & 1 + sumnest\ \ll 5, (9,13) \gg \\ = & 1 + (5 + sumnest\ (nest\ plus \ll (9,13) \gg)) \\ = & 1 + (5 + sumnest \ll 22 \gg) \\ = & 1 + (5 + (22 + sumnest\ (nest\ plus \ll\gg))) \\ = & 1 + (5 + (22 + sumnest \ll\gg)) \\ = & 1 + (5 + (22 + 0)) \\ = & 28 \end{array}$$

Compare this with the standard fold that sums a list.

$$
\begin{array}{ll}
 & \mathit{sumlist}\,[1,5,22] \\
= & 1 + \mathit{sumlist}\,[5,22] \\
= & 1 + (5 + \mathit{sumlist}\,[22]) \\
= & 1 + (5 + (22 + \mathit{sumlist}\,[\,])) \\
= & 1 + (5 + (22 + 0)) \\
= & 28
\end{array}
$$

To write *sumnest* in Haskell using *gfoldnest* we need the three terms $m\ a$ and $r\ (\mathit{Pair}\ a)$ and $r\ a$ all to match *Int*. Once again, higher-order unification would be needed to find the required bindings $m = \lambda x.\mathit{Int}$ and $r = \lambda x.\mathit{Int}$, so we make m and r match instead a type constructor *KInt* designed to mimic the functor K_{Int}.

$$\textbf{newtype}\ \mathit{KInt}\ a \quad = \quad \mathit{KInt}\ \mathit{Int}$$

To reinforce the difference between first-order and higher-order unification observe that $\mathit{KInt}\ a$ is isomorphic to *Int*, whereas $K_{\mathit{Int}}\ a$ is equal to *Int*. Now we can at least construct a function $\mathit{sumnest}'$ that has a type resembling that of *sumnest*.

$$
\begin{array}{lcl}
\mathit{zero}' & : & \mathit{KInt}\ a \\
\mathit{zero}' & = & \mathit{KInt}\ 0
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{plus}' & : & (\mathit{KInt}\ a, \mathit{KInt}\ a) \to \mathit{KInt}\ a \\
\mathit{plus}'\ (\mathit{KInt}\ x, \mathit{KInt}\ y) & = & \mathit{KInt}\ (x + y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sumnest}' & : & \mathit{Nest}\ (\mathit{KInt}\ a) \to \mathit{KInt}\ a \\
\mathit{sumnest}' & = & \mathit{gfoldnest}\ \mathit{zero}'\ \mathit{plus}'\ \mathit{plus}'
\end{array}
$$

To implement *sumnest* we wrap each integer in a *KInt* constructor, call $\mathit{sumnest}'$ and unwrap the result.

$$
\begin{array}{lcl}
\mathit{unKInt} & : & \mathit{KInt}\ a \to \mathit{Int} \\
\mathit{unKInt}\ (\mathit{KInt}\ x) & = & x
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{sumnest} & : & \mathit{Nest}\ \mathit{Int} \to \mathit{Int} \\
\mathit{sumnest} & = & \mathit{unKInt} \cdot \mathit{sumnest}' \cdot \mathit{nest}\ \mathit{KInt}
\end{array}
$$

3.1.4 Reductions

Summation on nests has type $Nest \cdot K_{Int} \dot{\to} K_{Int}$. It is one of a class of generalised folds known as reductions. A *reduction* is a generalised fold with type $T \cdot K_b \dot{\to} K_c$, for some b and c. (Meertens [Mee96] studies the special case where T is regular and $b = c$ and calls that a reduction.) We generalise *KInt*, which mimics the constant functor K_{Int}, to a curried binary type constructor K that can be used to mimic any constant functor.

$$\textbf{newtype}\ K\ a\ x \quad = \quad K\ a$$

Now the partial application $K\ a$ mimics the constant functor K_a.

The alternative to using the constructor K is to specialise the type signature of *gfoldnest* to give a reduction operator *rednest*. The specialisation replaces $m\ a$ by b and $r\ a$ and $r\ (Pair\ a)$ by c.

$$rednest \ :: \ c \to ((b,\ c) \to c) \to (Pair\ b \to b) \to Nest\ b \to c$$

$$\begin{aligned} &rednest\ nil\ cons\ bin\ NilN \ = \ nil \\ &rednest\ nil\ cons\ bin\ (ConsN\ (a, x)) \\ &\qquad = \ cons\ (a,\ rednest\ nil\ cons\ bin\ (nest\ bin\ x)) \end{aligned}$$

Now *sumnest* can be defined using the reduction operator as follows.

$$\begin{aligned} sumnest \ &:: \ Nest\ Int \to Int \\ sumnest \ &= \ rednest\ 0\ (uncurry\ (+))\ (uncurry\ (+)) \end{aligned}$$

We are interested in reductionsbecause the operator that produces them has such a simple type signature they also have much simpler fold-fusion laws, as we shall see.

3.1.5 Map-fusion laws

The map-fusion law for standard folds on lists is as follows:

$$foldlist\ f \cdot List\ k \quad = \quad foldlist\ (f \cdot Base\ (k, id))$$

The left-hand side applies k to each element of the list and then uses f to replace each constructor function. The law asserts that these two operations can always be interleaved in a more efficient single pass. The map-fusion law

for generalised folds on nests [BP99b], however, has a condition.

Map-fusion law for nests Given the typings

$$\begin{array}{rcl} f & : & Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R \\ g & : & Pair \cdot M \dot{\to} M \cdot Pair \\ g' & : & Pair \cdot M' \dot{\to} M' \cdot Pair \\ k & : & M' \dot{\to} M \end{array}$$

we have

$$(\!|f \mid g|\!)_{NestF} \cdot Nest\, k = (\!|f \cdot Base\ (k, id) \mid g'|\!)_{NestF} \Leftarrow g \cdot Pair\ k = k_{Pair} \cdot g'$$

Note that k_{Pair} has type $M' \cdot Pair \dot{\to} M \cdot Pair$ because k has type $M' \dot{\to} M$. To explain why the condition is necessary, we shall give, for the special case of nests, Bird and Paterson's derivation of the generic map-fusion law. By the universal property (and using the fact that *Nest* and *Base* are functors), we need only find conditions for

$$(\!|f \mid g|\!)_{NestF} \cdot Nest\, k \cdot \alpha \;=\; f \cdot Base\ (k, (\!|f \mid g|\!)_{NestF} \cdot Nest\ (k \cdot g'))$$

We reason

$$\begin{array}{ll} & (\!|f \mid g|\!)_{NestF} \cdot Nest\, k \cdot \alpha \\ = & \quad \{ \text{ naturality of } \alpha \ \} \\ & (\!|f \mid g|\!)_{NestF} \cdot \alpha \cdot Base\ (k, Nest\ (Pair\, k)) \\ = & \quad \{ \text{ definition of generalised fold; } Base \text{ and } Nest \text{ are functors } \} \\ & f \cdot Base\ (k, (\!|f \mid g|\!)_{NestF} \cdot Nest\ (g \cdot Pair\, k)) \end{array}$$

The condition is now needed to complete the derivation.

3.1.6 Fold-fusion laws

The fold-fusion law for standard folds on lists is as follows:

$$k \cdot foldlist\, f \;=\; foldlist\, f' \;\Leftarrow\; k \cdot f \;=\; f \cdot Base\ (id, k)$$

The condition says that a function k can be pushed through a function f, turning it into a function f'. The law says that if the condition is satisfied then the result of pushing k through the composition of f's produced by folding a list is a similar composition of f''s. The fold-fusion law for generalised

folds on nests [BP99b] has a condition for the auxiliary parameter as well.

Fold-fusion law for nests Given the typings

$$\begin{array}{rcl} f & : & Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R \\ f' & : & Base \cdot \langle M \cdot M', R' \cdot Pair \rangle \dot{\to} R' \\ g & : & Pair \cdot M \dot{\to} M \cdot Pair \\ g' & : & Pair \cdot M \cdot M' \dot{\to} M \cdot M' \cdot Pair \\ k & : & R \cdot M' \dot{\to} R' \end{array}$$

we have

$$k \cdot (\llparenthesis f \mid g \rrparenthesis_{NestF})_{M'} = \llparenthesis f' \mid g' \rrparenthesis_{NestF}$$

$$\Leftarrow \quad \exists p : \ k \cdot f = f' \cdot Base\ (id, k \cdot R\, p) \quad \text{and} \quad M\, p \cdot g = g'$$

The type required for p is $Pair \cdot M' \dot{\to} M' \cdot Pair$. It is hard to satisfy these conditions unless the terms $M\, p$ and $R\, p$ are eliminated by replacing them with identity functions. One way to do this is to let M and R and R' be constant functors.

Fold-fusion law for reductions on nests Given the typings

$$\begin{array}{rcl} f & : & Base\ (b, c) \to c \\ f' & : & Base'\ (b, c') \to c' \\ g & : & Pair\, b \to b \\ k & : & c \to c' \end{array}$$

we have

$$k \cdot \llparenthesis f \mid g \rrparenthesis_{NestF} = \llparenthesis f' \mid g \rrparenthesis_{NestF} \quad \Leftarrow \quad k \cdot f = f' \cdot Base\ (id, k)$$

Another way is to chose $p{=}id_{Pair}$ with $M'{=}Id$.

Specialised fold-fusion law for nests Given the typings

$$\begin{array}{rcl} f & : & Base \cdot \langle M, R \cdot Pair \rangle \stackrel{\cdot}{\to} R \\ f' & : & Base \cdot \langle M, R' \cdot Pair \rangle \stackrel{\cdot}{\to} R' \\ g & : & Pair \cdot M \stackrel{\cdot}{\to} M \cdot Pair \\ k & : & R \cdot M \stackrel{\cdot}{\to} R' \end{array}$$

we have

$$k \cdot (\!| f \mid g |\!)_{NestF} = (\!| f' \mid g |\!)_{NestF} \quad \Leftarrow \quad k \cdot f = f' \cdot Base\ (id, k)$$

The law for reductions is a special case of this, as is the following law for simple folds, obtained by taking $M{=}Id$ and $g{=}id_{Pair}$.

$$k \cdot (\!| f |\!)_{NestF} = (\!| f' |\!)_{NestF} \quad \Leftarrow \quad k \cdot f = f' \cdot Base\ (id, k)$$

It is unsurprising that this fold-fusion law resembles the fold-fusion law above for *foldlist*, given that the two fold operators have similar universal properties. There is no such correspondence for map-fusion laws: map-fusing a simple fold gives a generalised fold. This is also unsurprising, given that the map operators for nests and lists are not similar.

The reader is bound to wonder if all these fold-fusion laws will actually be used in this thesis. The first most general law will not be used; we quoted it from [BP99b] just so that we could derive the laws that follow it as special cases. The fourth law, which is for simple folds, will not be used as it was supplied to illustrate the close link between simple folds and standard folds.

The law for reductions is used in [BP99b] to derive a reduction for summing a nest given reductions for flattening a nest to a list and for summing a list. We shall use it ourselves to derive a similar law for efficient reductions (defined later) from which we prove the fold-equality law and the fold-equivalence law. The specialised fold-fusion law will be used to show that zips have a higher-order naturality property. The law for reductions cannot be used for this because zips are not always reductions.

3.2 Efficient folds for nests

Generalised folds and naive list reversal have something important in common. Both can be made more efficient by first specifying a generalisation and

then deriving a direct definition. The technique used is that of introducing an accumulating parameter. We shall review the technique as applied to the familiar problem of list reversal before applying it to generalised folds.

3.2.1 Improving on naive list reversal

Below is a naive program to reverse a list. It takes quadratic time in the length of the list [Bir98], as opposed to the ideal linear time.

$$\begin{array}{lcl} \mathit{reverse} & :: & [a] \rightarrow [a] \\ \mathit{reverse}\ [\,] & = & [\,] \\ \mathit{reverse}\ (x : \mathit{xs}) & = & (\mathit{reverse}\,\mathit{xs}) +\!\!+ [x] \end{array}$$

The cause of the inefficiency is the $(+\!\!+)$ operation, which takes linear time in the length of its left argument. We prefer to build up lists using the $(:)$ operation, which takes constant time. To achieve this, we specify a new function *revcat* that takes an extra parameter.

$$\mathit{revcat}\,\mathit{xs}\,\mathit{ys} \quad = \quad \mathit{reverse}\,\mathit{xs} +\!\!+ \mathit{ys}$$

This function is clearly a generalisation because

$$\mathit{reverse}\,\mathit{xs} \quad = \quad \mathit{revcat}\,\mathit{xs}\,[\,]$$

Now we can derive a direct recursive linear-time definition of *revcat*.

$$\begin{array}{lcl} \mathit{revcat}\ [\,]\,\mathit{ys} & = & \mathit{ys} \\ \mathit{revcat}\ (x : \mathit{xs})\,\mathit{ys} & = & \mathit{revcat}\,\mathit{xs}\ (x : \mathit{ys}) \end{array}$$

The parameter *ys* is called an accumulating parameter because it accumulates information as the function *revcat* recurses.

3.2.2 Improving on generalised folds

A generalised fold is inefficient because it recurses only after mapping over the remainder of the input structure. We shall remove this map to get a new fold called an efficient fold. We specify the efficient fold operator by introducing an accumulating parameter, h.

$$\{\![f \mid g \mid h]\!\}_{\mathit{NestF}} \quad = \quad (\![f \mid g]\!)_{\mathit{NestF}} \cdot \mathit{Nest}\,h$$

Now we derive a direct recursive definition, which will perform maps over pairs instead of over nests.

$$
\begin{array}{ll}
& \{\![f \mid g \mid h]\!\}_{NestF} \cdot \alpha \\
= & \quad \{ \text{ specification of efficient fold } \} \\
& (\![f \mid g]\!)_{NestF} \cdot Nest\ h \cdot \alpha \\
= & \quad \{ \text{ naturality of } \alpha \ \} \\
& (\![f \mid g]\!)_{NestF} \cdot \alpha \cdot Base\ (h, Nest\ (Pair\, h)) \\
= & \quad \{ \text{ definition of generalised fold } \} \\
& f \cdot Base\ (id, (\![f \mid g]\!)_{NestF} \cdot Nest\ g) \cdot Base\ (h, Nest\ (Pair\, h)) \\
= & \quad \{ \ Base \text{ and } Nest \text{ are functors } \} \\
& f \cdot Base\ (h, (\![f \mid g]\!)_{NestF} \cdot Nest\ (g \cdot Pair\, h)) \\
= & \quad \{ \text{ specification of efficient fold } \} \\
& f \cdot Base\ (h, \{\![f \mid g \mid g \cdot Pair\, h]\!\}_{NestF})
\end{array}
$$

The definition that we have just derived for the efficient fold operator can be strengthened to a universal property. We shall sketch the proof of this from [BGM] at the end of this chapter.

$$
\begin{array}{rcl}
\{\![-|-|-]\!\}_{NestF} & : & (Base \cdot \langle M, R \cdot Pair \rangle \dot{\to} R) \to \\
& & (Pair \cdot M \dot{\to} M \cdot Pair) \to \\
& & (M' \dot{\to} M) \to \\
& & (Nest \cdot M' \dot{\to} R)
\end{array}
$$

$$
\chi\, h \ = \ \{\![f \mid g \mid h]\!\}_{NestF} \ \equiv \ \chi\, h \cdot \alpha \ = \ f \cdot Base\ (h,\ \chi\ (g \cdot Pair\, h))
$$

3.2.3 Efficient summation on nests

We shall call the special case where M and M' and R in the universal property are all constant functors, an *efficient reduction*. Remember how to sum a nest with a reduction:

$$
\begin{array}{rcl}
sumnest & : & Nest\, Int \to Int \\
sumnest & = & (\![kzero, plus \mid plus]\!)_{NestF}
\end{array}
$$

We can use an efficient reduction instead:

$$\begin{array}{lcl} \mathit{esumnest} & : & \mathit{Nest\,Int} \to \mathit{Int} \\ \mathit{esumnest} & = & \{\![\mathit{kzero}, \mathit{plus} \mid \mathit{plus} \mid \mathit{id}]\!\}_{\mathit{NestF}} \end{array}$$

In the sample evaluation below, let $\mathit{esumnest}'$ be the partial application $\{\![\mathit{kzero}, \mathit{plus} \mid \mathit{plus} \mid -]\!\}_{\mathit{NestF}}$ and let h be defined by $h_0 = \mathit{id}$ and $h_{i+1} = (+) \cdot \mathit{Pair}\, h_i$.

$$\begin{array}{cl} & \mathit{esumnest} \ll 1, (2,3), ((4,5),(6,7)) \gg \\ = & \mathit{esumnest}'\ h_0 \ll 1, (2,3), ((4,5),(6,7)) \gg \\ = & h_0\ 1 + \mathit{esumnest}'\ h_1 \ll (2,3), ((4,5),(6,7)) \gg \\ = & 1 + (h_1\ (2,3) + \mathit{esumnest}'\ h_2 \ll ((4,5),(6,7)) \gg) \\ = & 1 + (5 + (h_2\ ((4,5),(6,7)) + \mathit{esumnest}'\ h_3 \ll\gg)) \\ = & 1 + (5 + (22 + 0)) \\ = & 28 \end{array}$$

Contrast this with how *sumnest* behaved on the same input.

$$\begin{array}{cl} & \mathit{sumnest} \ll 1, (2,3), ((4,5),(6,7)) \gg \\ = & 1 + \mathit{sumnest}\ (\mathit{nest\,plus} \ll (2,3), ((4,5),(6,7)) \gg) \\ = & 1 + \mathit{sumnest} \ll 5, (9,13) \gg \\ = & 1 + (5 + \mathit{sumnest}\ (\mathit{nest\,plus} \ll (9,13) \gg)) \\ = & 1 + (5 + \mathit{sumnest} \ll 22 \gg) \\ = & 1 + (5 + (22 + \mathit{sumnest}\ (\mathit{nest\,plus} \ll\gg))) \\ = & 1 + (5 + (22 + \mathit{sumnest} \ll\gg)) \\ = & 1 + (5 + (22 + 0)) \\ = & 28 \end{array}$$

Observe that *esumnest* takes just a single pass through the whole input. The head of what remains of the input has type $\mathit{Pair}^k\,\mathit{Int}$ for some k and it is reduced by h_k to a single integer. In contrast, *sumnest* performs a map every time a *ConsN* constructor is removed, so it is always applied to nests that have a completely reduced head. As it happens, these two evaluations are equally efficient and they were chosen for clarity. Two similar evaluations that do motivate efficient folds are provided by Hinze [Hin99a] for the different datatype of de Bruijn terms [BP99a]. We shall summarise Hinze's argument in Section 3.4 once we have generalised our fold operators to linear datatypes.

3.2.4 Efficient folds in Haskell

Now we write the efficient fold operator in Haskell. It turns out that we can only implement the efficient fold operator for a special case where the argument being mapped has the type $M' \cdot K_{a'} \dot{\to} M \cdot K_a$ rather than the more general $M' \dot{\to} M$. The problem, explained in [Hin99a], is that the operator is defined using polymorphic recursion on the level of functors. We do not know of any languages where this is allowed. By introducing a constant functor, we shift the polymorphic recursion back to the level of types, because the type of the fold returned by the operator must now be $\mathit{Nest} \cdot M' \cdot K_{a'} \dot{\to} R \cdot K_a$. Since neither of these types are collections of arrows, we write them in Haskell without type signatures.

$$
\begin{aligned}
\mathit{efoldnest} \;::\; & \mathbf{forall}\, a\; m\; m'\; r. \\
& (\mathbf{forall}\, b.\, r\; b) \to \\
& (\mathbf{forall}\, b.\, (m\; b, r\; (\mathit{Pair}\; b)) \to r\; b) \to \\
& (\mathbf{forall}\, b.\, \mathit{Pair}\; (m\; b) \to m\; (\mathit{Pair}\; b)) \to \\
& (m'\; a' \to m\; a) \to \\
& \mathit{Nest}\; (m'\; a') \to r\; a
\end{aligned}
$$

$$\mathit{efoldnest\; nil\; cons\; bin\; tip\; NilN} = \mathit{nil}$$

$$
\begin{aligned}
& \mathit{efoldnest\; nil\; cons\; bin\; tip}\; (\mathit{ConsN}\; (a, x)) \\
& \quad = \mathit{cons}\; (\mathit{tip}\; a,\; \mathit{efoldnest\; nil\; cons}\; (\mathit{bin} \cdot \mathit{pair\; tip})\; \mathit{bin}\; x)
\end{aligned}
$$

Now it is the type a that changes during recursion, rather than the type constructor m'. The function *gfoldnest* can be defined using *efoldnest*.

$$\mathit{gfoldnest\; nil\; cons\; bin} \;=\; \mathit{efoldnest\; nil\; cons\; bin\; id}$$

The type signature above can be made more general by replacing the type $m'\; a'$ with a type variable a''. However, we do not know of any extra applications for this more general operator.

3.3 Fold-equality law

3.3.1 Motivation for fold-equality law

Chapter 2 told us how to flatten a nest with a simple fold. The type of this simple fold can be written as $\mathit{Nest} \cdot K_a \dot{\to} K_{[a]}$. It is unsurprising, therefore,

that a nest can also be flattened using an efficient reduction. We would like to deduce the parameters needed to do this from the parameter that is used by the simple fold operator. This section derives a fold-equality law that will enable us to do just that.

We cannot expect all simple folds to be efficient reductions. A counterexample is the identity function on nests. One of the parameters to the efficient reduction operator has a type $Pair\ b \to b$ for some b, that requires the fold to throw away information. Furthermore, not all efficient reductions are simple folds; consider *esumnest*. So the fold equality law must depend on a condition, to be derived, that relates the parameters of the two folds.

3.3.2 Derivation of fold-equality law

We want to find conditions on f, g, h and f' such that

$$\{\![f \mid g \mid h]\!\}_{NestF} \quad = \quad (\![f']\!)_{NestF}$$

This problem looks like a straightforward application of map-fusion for generalised folds. However, it is clear from the typing of the map-fusion law for reductions that the result of map-fusing a reduction is always a reduction and never a simple fold. Instead we must apply the universal property of simple folds.

$$\begin{array}{ll} & \{\![f \mid g \mid h]\!\}_{NestF} = (\![f']\!)_{NestF} \\ \equiv & \quad \{ \text{ universal property of simple folds } \} \\ & \{\![f \mid g \mid h]\!\}_{NestF} \cdot \alpha = f' \cdot Base\ (id, \{\![f \mid g \mid h]\!\}_{NestF}) \\ \equiv & \quad \{ \text{ definition of efficient folds } \} \\ & f \cdot Base\ (h, \{\![f \mid g \mid g \cdot Pair\ h]\!\}_{NestF}) = f' \cdot Base\ (id, \{\![f \mid g \mid h]\!\}_{NestF}) \end{array}$$

This last equation is true if the following conditions hold:

$$\begin{array}{rcl} f' & = & f \cdot Base\ (h, k) \\ k \cdot \{\![f \mid g \mid h]\!\}_{NestF} & = & \{\![f \mid g \mid g \cdot Pair\ h]\!\}_{NestF} \end{array}$$

The second of these conditions is an application of a fold-fusion law for efficient reductions on nests, which we now derive.

$$\begin{array}{ll} & k \cdot \{\![f \mid g \mid h]\!\}_{NestF} \\ = & \quad \{ \text{ specification of efficient folds } \} \\ & k \cdot (\![f \mid g]\!)_{NestF} \cdot Nest\ h \end{array}$$

$$
\begin{aligned}
= & \quad \{ \text{ fold-fusion: } \textbf{assume } k \cdot f = f' \cdot \mathit{Base}\ (k', k) \ \} \\
& (\!| f' \cdot \mathit{Base}\ (k', id) \mid g |\!)_{\mathit{NestF}} \cdot \mathit{Nest}\, h \\
= & \quad \{ \text{ map-fusion: } \textbf{assume } g' \cdot \mathit{Pair}\, k' = k' \cdot g \ \} \\
& (\!| f' \mid g' |\!)_{\mathit{NestF}} \cdot \mathit{Nest}\, k' \cdot \mathit{Nest}\, h \\
= & \quad \left\{ \begin{array}{l} \text{specification of efficient folds;} \\ \mathit{Nest} \text{ is a functor; } \textbf{assume } k' \cdot h = h' \end{array} \right\} \\
& \{\![f' \mid g' \mid h']\!\}_{\mathit{NestF}}
\end{aligned}
$$

This derivation is specialised to reductions so the fold-fusion law is used with simple conditions. In summary, we have

Fold-fusion law for efficient reductions on nests

$$k \cdot \{\![f \mid g \mid h]\!\}_{\mathit{NestF}} = \{\![f' \mid g' \mid h']\!\}_{\mathit{NestF}}$$

$$
\begin{aligned}
\Leftarrow \exists k' : \quad & k \cdot f = f' \cdot \mathit{Base}\ (k', k) \quad \text{and} \\
& g' \cdot \mathit{Pair}\, k' = k' \cdot g \quad \text{and} \\
& h' = k' \cdot h
\end{aligned}
$$

We can now immediately finish our derivation of the fold-equality law.

Fold-equality law for nests

$$\{\![f \mid g \mid h]\!\}_{\mathit{NestF}} = (\!| f \cdot \mathit{Base}\ (h, k) |\!)_{\mathit{NestF}}$$

$$
\begin{aligned}
\Leftarrow \exists k' : \quad & k \cdot f = f \cdot \mathit{Base}\ (k', k) \quad \text{and} \\
& g \cdot \mathit{Pair}\, k' = k' \cdot g \quad \text{and} \\
& g \cdot \mathit{Pair}\, h = k' \cdot h
\end{aligned}
$$

We shall soon use this law to rewrite a simple fold as an efficient reduction. An example in Chapter 8 sees us doing the reverse, indicating that the programmer can choose between these two different types of folds according to their pros and cons, as follows. Simple folds resemble familiar standard folds in that the only parameters needed are the replacements for the constructors *NilN* and *ConsN*. That is why we initially chose to flatten a nest with a simple fold. However, the novice programmer will find rank two type signatures difficult to use, especially when constructor functions are needed to enforce

bindings. Efficient reductions have simpler type signatures but as they do more than just replace constructor functions, it can be sometimes difficult to visualise their behaviour.

3.3.3 Application: Flattening on nests

Here, once again, is the simple fold that flattens a nest.

$$\begin{aligned} \mathit{hflatnest} &:: \mathit{Nest}\ a \to [a] \\ \mathit{hflatnest} &= \mathit{hfoldnest}\,[\,]\ (\lambda(a,x) \to a : \mathit{flatlp}\,x) \end{aligned}$$

$$\begin{aligned} \mathit{flatlp} &:: [\mathit{Pair}\ a] \to [a] \\ \mathit{flatlp} &= \mathit{concat} \cdot \mathit{map}\ (\backslash(y,z) \to [y,z]) \end{aligned}$$

In order to write *hflatnest* in the notation of category theory, let $\mathit{cons} : \mathit{Id} \times \mathit{List} \dot{\to} \mathit{List}$ and $\mathit{cat} : \mathit{Pair} \cdot \mathit{List} \dot{\to} \mathit{List}$ denote the uncurried versions of $(:)$ and $(+\!\!+)$ respectively. Let $\mathit{knil} : K_1 \dot{\to} \mathit{List}$ and $\mathit{wrap} : \mathit{Id} \dot{\to} \mathit{List}$ be defined by $\mathit{knil}\,x = [\,]$ and $\mathit{wrap}\,x = [x]$. Now we write

$$\begin{aligned} \mathit{hflatnest} &: \mathit{Nest} \dot{\to} \mathit{List} \\ \mathit{hflatnest} &= (\![\mathit{knil}, \mathit{cons} \cdot (\mathit{id} \times \mathit{flatlp})]\!)_{\mathit{NestF}} \end{aligned}$$

Here, the function *flatlp* flattens a list of pairs to a list. Here are three point-free properties of *flatlp* that are obviously correct when written pointwise.

$$\begin{aligned} \mathit{flatlp} \cdot \mathit{knil} &= \mathit{knil} \\ \mathit{flatlp} \cdot \mathit{wrap} &= \mathit{cat} \cdot (\mathit{wrap} \times \mathit{wrap}) \\ \mathit{flatlp} \cdot \mathit{cat} &= \mathit{cat} \cdot (\mathit{flatlp} \times \mathit{flatlp}) \end{aligned}$$

Although these three properties form an inductive definition, as a list must be an empty list, a singleton list or a concatenation of two lists, only the first two are valid Haskell when written pointwise.

$$\begin{aligned} & [\mathit{knil}, \mathit{cons} \cdot (\mathit{id} \times \mathit{flatlp})] \\ = \quad & \{\ x : xs = [x] +\!\!+ xs\ \} \\ & [\mathit{knil}, \mathit{cat} \cdot (\mathit{wrap} \times \mathit{flatlp})] \\ = \quad & \{\ \text{fusion laws of coproduct}\ \} \\ & [\mathit{knil}, \mathit{cat}] \cdot (\mathit{id}_1 + (\mathit{wrap} \times \mathit{flatlp})) \end{aligned}$$

$$= \quad \left\{ \text{ definition of } \mathit{Base} \right\}$$
$$[\mathit{knil}, \mathit{cat}] \cdot \mathit{Base}\ (\mathit{wrap}, \mathit{flatlp})$$

According to the fold-equality law, the corresponding efficient fold is

$$\begin{array}{lcl} \mathit{eflatnest} & : & \mathit{Nest} \dot{\to} \mathit{List} \\ \mathit{eflatnest} & = & \{\![\mathit{knil}, \mathit{cat} \mid g \mid \mathit{wrap}]\!\}_{\mathit{NestF}} \end{array}$$

The function g is given by the conditions of the law. For some k', we have

$$\begin{array}{rcl} \mathit{flatlp} \cdot [\mathit{knil}, \mathit{cat}] & = & [\mathit{knil}, \mathit{cat}] \cdot \mathit{Base}\ (k', \mathit{flatlp}) \\ g \cdot \mathit{Pair}\ k' & = & k' \cdot g \\ g \cdot \mathit{Pair}\ \mathit{wrap} & = & k' \cdot \mathit{wrap} \end{array}$$

Once we separate out the coproduct and take $g = \mathit{cat}$ and $k' = \mathit{flatlp}$, these conditions are exactly the three properties we gave for *flatlp*. In conclusion, the efficient fold that flattens a nest is

$$\begin{array}{lcl} \mathit{eflatnest} & : & \mathit{Nest} \dot{\to} \mathit{List} \\ \mathit{eflatnest} & = & \{\![\mathit{knil}, \mathit{cat} \mid \mathit{cat} \mid \mathit{wrap}]\!\}_{\mathit{NestF}} \end{array}$$

In Haskell, we write

$$\begin{array}{lcl} \mathit{eflatnest} & :: & \mathit{Nest}\ a \to [a] \\ \mathit{eflatnest} & = & \mathit{erednest}\, [\,]\ (\mathit{uncurry}\ (+\!\!+))\ (\mathit{uncurry}\, (+\!\!+))\ (:[\,]) \end{array}$$

Here, *erednest* is *efoldnest* with a specialised type signature.

$$\mathit{erednest} \quad :: \quad c \to ((b, c) \to c) \to (a \to b) \to (\mathit{Pair}\ b \to b) \to \mathit{Nest}\ a \to c$$

$$\mathit{erednest}\ \mathit{nil}\ \mathit{cons}\ \mathit{bin}\ \mathit{tip}\ \mathit{NilN}\ = \ \mathit{nil}$$
$$\mathit{erednest}\ \mathit{nil}\ \mathit{cons}\ \mathit{bin}\ \mathit{tip}\ (\mathit{ConsN}\ (a, x))$$
$$= \ \mathit{cons}\ (\mathit{tip}\ a,\ \mathit{erednest}\ \mathit{nil}\ \mathit{cons}\ (\mathit{bin} \cdot \mathit{pair}\ \mathit{tip})\ \mathit{bin}\ x)$$

3.3.4 Comparisons of flattening functions

We now contrast *eflatnest* with *hflatnest* by applying them both to the same input.

$$\mathit{hflatnest}\ \ll 1, (2,3), ((4,5),(6,7)) \gg$$
$$= \ 1 : \mathit{flatlp}\ ((2,3) : \mathit{flatlp}\ (((4,5),(6,7)) : \mathit{flatlp} \ll\gg))$$
$$= \ 1 : \mathit{flatlp}\ ((2,3) : \mathit{flatlp}\ (((4,5),(6,7)) : [\,]))$$

$$
\begin{aligned}
&= 1 : \mathit{flatlp}\ ((2,3) : \mathit{flatlp}\ [((4,5),(6,7))]) \\
&= 1 : \mathit{flatlp}\ [(2,3),(4,5),(6,7)] \\
&= [1,2,3,4,5,6,7]
\end{aligned}
$$

As with the summation example, let *eflatnest′* be the partial application of *eflatnest* and let h be defined by $h_0 = \mathit{wrap}$ and $h_{i+1} = \mathit{cat} \cdot \mathit{Pair}\ h_i$.

$$
\begin{aligned}
& \mathit{eflatnest}\ \ll 1,(2,3),((4,5),(6,7)) \gg \\
= \ & \mathit{eflatnest}'\ h_0\ \ll 1,(2,3),((4,5),(6,7)) \gg \\
= \ & h_0\ 1 +\!\!+\ \mathit{eflatnest}'\ h_1\ \ll (2,3),((4,5),(6,7)) \gg \\
= \ & [1] +\!\!+ (h_1\ (2,3) +\!\!+\ \mathit{eflatnest}'\ h_2\ \ll ((4,5),(6,7)) \gg) \\
= \ & [1] +\!\!+ ([2,3] +\!\!+ (h_2\ ((4,5),(6,7)) +\!\!+\ \mathit{eflatnest}'\ h_3 \ll\gg)) \\
= \ & [1] +\!\!+ ([2,3] +\!\!+ ([4,5,6,7] +\!\!+ [\,])) \\
= \ & [1,2,3,4,5,6,7]
\end{aligned}
$$

Observe that *hflatnest* turns lists of pairs of pairs into lists of pairs and then into lists, whereas *eflatnest* turns integers or pairs of integers or even pairs of pairs of integers straight into lists for concatenation.

3.4 Folds for linear datatypes

In Chapter 2, we wrote the simple fold operator for nests entirely in terms of *NestF* and *Nest*. We could then generalise the simple fold operator to any nested datatype, simply by abstracting over *Nest* and *NestF*. In contrast, the generalised fold operator for nests is defined in terms of *Base* and *Pair* and *Nest* so we can only generalise it to functors of the form

$$
\begin{aligned}
T &\approx F\,T \\
F\,X &= B \cdot \langle \mathit{Id}, X \cdot Q \rangle
\end{aligned}
$$

This form is nevertheless general enough to include all regular datatypes. We can use it to define folds for and reason about a wide variety of datatypes without using case analysis. However, the form is not general enough to include all linear datatypes as some datatypes can recurse in several different ways. For example, the datatype below models the de Bruijn notation for writing lambda terms without using variables. Each bound variable is

represented by a term $Var\,Succ^n\,Zero$ where n indicates the distance of the binding lambda abstraction.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Term}\ a & = & \mathit{Var}\ a \mid \mathit{App}\ (\mathit{Term}\ a, \mathit{Term}\ a) \mid \mathit{Lam}\ (\mathit{Term}\ (\mathit{Incr}\ a)) \\
\textbf{data}\ \mathit{Incr}\ a & = & \mathit{Zero} \mid \mathit{Succ}\ a
\end{array}
$$

For example, the lambda term $\lambda xyz.xz$ is represented by

$$
\mathit{Lam}\ (\mathit{Lam}\ (\mathit{Lam}\ (\mathit{App}\ (\mathit{Var}\ (\mathit{Succ}\ (\mathit{Succ}\ \mathit{Zero})), \mathit{Var}\ \mathit{Zero}))))
$$

Free variables can be represented by elements of the type a. To handle all linear datatypes we need the hofunctor F to have the form

$$
F\,X \quad = \quad B \cdot \langle \mathit{Id}, X \cdot P_1, \ldots, X \cdot P_n \rangle
$$

Here, B denotes a polynomial n-ary functor; if $n = 0$ then T is polynomial but this case does not interest us. We shall see shortly that this form is not canonical. We shall avoid ellipses when defining generic operations by assuming that $n = 1$, because our reasoning can always be generalised to larger n. In other words, we have traded rigour for generality.

We can generalise to linear datatypes the universal property of the generalised fold operator.

$$
h = (\!|\, f \mid g \,|\!)_F \quad \equiv \quad h \cdot \alpha = f \cdot B\ (\mathit{id}, h \cdot T\ g)
$$

In general, if B is an n-ary functor (for $n \geq 2$) then the operator will take n parameters. Now we can see that our form for hofunctors is not canonical. The type of de Bruijn terms can be written in category notation as

$$
\begin{array}{rcl}
\mathit{DeB} & \approx & \mathit{DeBF}\ \mathit{DeB} \\
\mathit{DeBF}\ X & = & \mathit{Id} + X \times X + X \cdot \mathit{Incr} \\
\mathit{Incr} & = & K_1 + \mathit{Id}
\end{array}
$$

Now DeBF can be put in our special form for hofunctors in two different ways, DeBF_1 and DeBF_2 below.

$$
\begin{array}{rcl}
\mathit{DeBF}_1\ X & = & \mathit{BaseDB}_1 \cdot \langle \mathit{Id}, X, X \cdot \mathit{Incr} \rangle \\
\mathit{DeBF}_2\ X & = & \mathit{BaseDB}_2 \cdot \langle \mathit{Id}, X, X, X \cdot \mathit{Incr} \rangle
\end{array}
$$

The base functors are defined by

$$
\begin{array}{rcl}
\mathit{BaseDB}_1\ (X, Y, Z) & = & X + Y \times Y + Z \\
\mathit{BaseDB}_2\ (X, Y, Y', Z) & = & X + Y \times Y' + Z
\end{array}
$$

The generalised fold operator for the type of de Bruijn terms can have either three or four parameters. However, the sets of parameters have the same types and once the base functors have been substituted for, the two operators are the same, except for one being an instance of the other.

Hinze [Hin99a] shows that generalised folds take quadratic time for the datatype of de Bruijn terms. A map over the structure $Lam^n\ (Var\ (Succ^n\ a))$ takes linear time and that is why a generalised fold over the structure, with a map at each recursive step, will take quadratic time.

We generalise the universal property of the efficient fold operator to

$$\chi\, h \;=\; \{\![f \mid g \mid h]\!\}_F \;\equiv\; \chi\, h \cdot \alpha \;=\; f \cdot B\ (h,\ \chi\ (g \cdot Q\, h))$$

The map-fusion law generalises to

$$(\![f \mid g]\!)_F \cdot T\, k = (\![f \cdot B\ (k, id) \mid g']\!)_F \;\Leftarrow\; g \cdot Q\ k = k_Q \cdot g'$$

The fold-fusion law generalises to

$$k \cdot ((\![f \mid g]\!)_F)_{M'} = (\![f' \mid g']\!)_F$$

$$\Leftarrow \quad \exists p :\ k \cdot f = f' \cdot B\ (id, k \cdot R\, p) \quad \text{and} \quad M\ p \cdot g = g'$$

We can even generalise the fold-equality law.

$$\{\![f \mid g \mid h]\!\}_F = (\![f \cdot B\ (h, k)]\!)_F$$

$$\begin{aligned} \Leftarrow \exists k' :\ & k \cdot f = f \cdot B\ (k', k) \quad \text{and} \\ & g \cdot Q\, k' = k' \cdot g \quad \text{and} \\ & g \cdot Q\, h = k' \cdot h \end{aligned}$$

The type of the efficient fold operator specialised to efficient reductions is as follows:

$$\begin{aligned} \{\![-|-|-]\!\}_F \;:\; & (B\ (b, c) \to c) \to (Q\ b \to b) \to (a \to b) \to \\ & T\ a \to c \end{aligned}$$

When T is regular then $Q = Id$ and we can use the generalised fold operator to define the standard fold operator.

$$\begin{aligned} fold_F \quad & : \quad (B\ (b, c) \to c) \to T\ b \to c \\ fold_F\ f \quad & = \quad (\![f \mid id_b]\!)_F \end{aligned}$$

3.4.1 Alternating lists

We shall only consider single-parameter first-order datatypes for the rest of this thesis. However, to help the interested reader generalise the thesis to multi-parameter datatypes, we shall give the generalised fold operator for alternating lists. Recall the definition of alternating lists in category notation.

$$\begin{aligned} \mathit{ALF}\, X &= \mathit{BaseAL} \cdot \langle \mathit{Id}, X \cdot \mathit{Swap} \rangle \\ \mathit{BaseAL} &= \mathit{KK}_1 + \mathit{Outl} \cdot \mathit{Outl} \times \mathit{Outr} \\ \mathit{Swap} &= \langle \mathit{Outr}, \mathit{Outl} \rangle \end{aligned}$$

The generalised fold operator is as follows.

$$\begin{aligned} (\!|-|-|\!)_{\mathit{NestF}} \;:\; & (\mathit{BaseAL} \cdot \langle M, R \cdot \mathit{Swap} \rangle \dot{\to} R) \to \\ & (\mathit{Swap} \cdot M \dot{\to} M \cdot \mathit{Swap}) \to \\ & (\mathit{AL} \cdot (M \times N) \dot{\to} R) \end{aligned}$$

$$(\!|f \mid g|\!)_{\mathit{ALF}} \cdot \alpha = f \cdot \mathit{BaseAL}\ (\mathit{id},\ (\!|f \mid g|\!)_{\mathit{ALF}} \cdot \mathit{AL}\, g)$$

In Haskell, this is

$$\begin{aligned} \mathit{gfoldal} \;::\; & \mathbf{forall}\ c\ d\ r. \\ & (\mathbf{forall}\ a\ b.\ r\ a\ b) \to \\ & (\mathbf{forall}\ a\ b.\ (m\ a, r\ b\ a) \to r\ a\ b) \to \\ & (\mathbf{forall}\ b.n\ b \to m\ b) \to \\ & (\mathbf{forall}\ a.m\ a \to n\ a) \to \\ & \mathit{AL}\ (m\ c)\ (n\ d) \to r\ c\ d \end{aligned}$$

$$\begin{aligned} & \mathit{gfoldal}\ \mathit{nil}\ \mathit{cons}\ \mathit{nb}\ \mathit{ma}\ \mathit{NilAL} = \mathit{nil} \\ & \mathit{gfoldal}\ \mathit{nil}\ \mathit{cons}\ \mathit{nb}\ \mathit{ma}\ (\mathit{ConsAL}\ (a, x)) \\ & \qquad = \mathit{cons}\ (a,\ \mathit{gfoldal}\ \mathit{nil}\ \mathit{cons}\ \mathit{nb}\ \mathit{ma}\ (\mathit{al}\ \mathit{nb}\ \mathit{ma}\ x)) \end{aligned}$$

3.4.2 Square Matrices

Now let us consider higher-order datatypes. Generalised folds for square matrices have the type

$$\mathit{SM}\ (m\, h)\ a \to n\ h\ a$$

The generalised fold operator must therefore return a function of this type. The map operator for SM must be supplied with a map for $m\, h$, for every

h. To ensure this we could give the type signature the following type class context, but it is not legal Haskell:

$$\textbf{forall}\ h.\ \mathit{Functor}\ (m\,h)$$

We need some way of declaring m to be a higher-order functor that takes functors to functors but it is not clear how to do this. Clare Martin has shown that the desired type signature can be specialised to a reduction with a legal class context by taking m and n to be constant hofunctors but the resulting operator does not seem to be of any use. The same seems to be true of the general case when m and n are not constant hofunctors for the same reasons as before with the simple fold operator [Mar01].

3.5 Proofs of universal properties for folds

In [BP99b], Bird and Paterson prove the following theorem, two special cases of which are the universal properties of generalised folds and efficient folds.

Theorem on the uniqueness of arrows
Suppose that F is an endofunctor on $\mathbf{C}$ with an initial algebra $\alpha : F\ T \rightarrow T$ and that Ψ is a natural transformation with typing

$$\Psi_A \quad : \quad (L\,A \rightarrow B) \rightarrow (L\ (F\ A) \rightarrow B)$$

where $L : \mathbf{C} \rightarrow \mathbf{D}$ for some category $\mathbf{D}$. Then there is a unique $x : L\ T \rightarrow B$ such that

$$x \cdot L\alpha \quad = \quad \Psi\,x$$

(Note that in the type of Ψ_A the inner and the outer arrows belong to different categories.) We shall sketch Bird and Paterson's proof of this theorem shortly but first we shall demonstrate that the universal properties of this chapter are both instances of the theorem. We consider only linear T so we define F by

$$F\,X \quad = \quad B' \cdot \langle \mathit{Id}, X \cdot Q \rangle$$

The extension to non-linear datatypes is trivial.

3.5.1 Uniqueness of generalised folds

Suppose that $\mathbf{C} = \mathbf{D} = \mathbf{Coc}\ (\mathbf{Fun})$ and L is defined on functors by $L\,F = F \cdot M$ and on natural transformations by $L\,\beta = \beta_M$. If Ψ is defined by $\Psi\,h = f \cdot B'\ (id, h \cdot T\,g)$ for appropriate parameters f and g then according to the theorem, there is a unique arrow $(\!|f \mid g|\!)_F : T \cdot M \dot{\to} R$ such that

$$(\!|f \mid g|\!)_F \cdot (\alpha_F)_M \quad = \quad f \cdot B'\ (id, (\!|f \mid g|\!)_F \cdot T\,g)$$

This confirms the right-to-left direction of the universal property; the left-to-right direction follows immediately from the definition of $(\!|f \mid g|\!)_F$.

3.5.2 Uniqueness of efficient folds

The instantiation for efficient folds is more complex, because a unique *mapping* on arrows is required, which must be a unique arrow in a higher order category $\mathbf{D}$, the details of which are yet to be determined. Let $\mathbf{C} = \mathbf{Coc}\ (\mathbf{Fun})$ as before. We want to show that there exists a unique arrow χ in $\mathbf{D}$ that satisfies the definition of $\{\!|f \mid g \mid -|\!\}$. Its type should be

$$(M' \dot{\to} M) \to (T \cdot M' \dot{\to} R)$$

Following the proof in [BGM], we shall broaden this type to

$$(W \dot{\to} M \cdot X) \to (T \cdot W \dot{\to} R \cdot X)$$

In fact, χ must be a collection of such arrows indexed by functor pairs (W, X).

$$\chi_{W,X} \quad : \quad (W \dot{\to} M \cdot X) \to (T \cdot W \dot{\to} R \cdot X)$$

For χ to be unique by the theorem, it must be an arrow in $\mathbf{D}$ of type $L\,T \to B$. This information alone is enough to fix for us the objects and arrows of $\mathbf{D}$. An object U of $\mathbf{D}$ must map a pair of endofunctors (W, X) to a single endofunctor $U\ (W, X)$. An arrow $\xi : U \to V$ of $\mathbf{D}$ must map a pair of endofunctors (W, X) to a total function of type

$$\xi_{W,X} \quad : \quad (W \dot{\to} M \cdot X) \to (U\ (W, X) \dot{\to} V\ (W, X))$$

When $U = L\,A$ and $V = B$ we have,

$$\xi_{W,X} \quad : \quad (W \dot{\to} M \cdot X) \to (L\,A\ (W, X) \dot{\to} B\ (W, X))$$

Clearly, χ is an arrow of $\mathbf{D}$ because the types of $\xi_{W,X}$ and $\chi_{W,X}$ match with $A = T$ and $L\,A\ (W, X) = A \cdot W$ and $B\ (W, X) = R \cdot X$. Now χ is to be the unique solution to

$$\chi \circ P\,\alpha = \Psi\ \chi$$

Here, $\circ$ denotes the composition of arrows of $\mathbf{D}$. In fact, we really need

$$(\chi \circ P\,\alpha)_{W,X}\ p \quad = \quad (\Psi\,\chi)_{W,X}\ p$$

To make this equality on arrows of $\mathbf{D}$ match the equality on arrows of $\mathbf{Coc}$ ($\mathbf{Fun}$) that defines $\{\![f \mid g \mid -]\!\}_F$ we define $\circ$ as follows.

$$(\xi' \circ \xi'')\ p \quad = \quad \xi'\ p \circ \xi''\ p$$

Now we choose

$$P\ \eta\ (W, X)\ p \quad = \quad \eta_W$$

Finally, for appropriate parameters f and g and h we define Ψ by

$$(\Psi\,\xi)_{W,X}\ h \quad = \quad f \cdot B'\ (h, \xi\ (g \cdot Q\,h))$$

These substitutions give us the required equation.

$$(\{\![f \mid g \mid -]\!\}_F\ h) \cdot (\alpha_F)_W \quad = \quad f \cdot B'\ (h, \{\![f \mid g \mid -]\!\}_F\ (g \cdot Q\,h))$$

Since Ψ takes arrows

$$(W \dot{\to} M \cdot X) \to (T \cdot W \dot{\to} R \cdot X)$$

to arrows

$$(W \dot{\to} M \cdot X) \to (F\ T \cdot W \dot{\to} R \cdot X)$$

we see that it is a natural transformation of the right type.

$$\Psi_A \ \ :\ \ (P\ A \to B) \to (P\ (F\ A) \to B)$$

This completes the proof of the universal property of efficient folds.

3.5.3 Proof sketch for the theorem

Bird and Paterson give two proofs of their theorem. The first proof exploits the details of how T is constructed as a least fixed point of F using colimits. It assumes that F and L preserve colimits and that L preserves initiality. Both these conditions are satisfied in our uses of the theorem. The proof establishes both the uniqueness and the existence of a solution to the equation. It takes the form of an induction on the number used to index the arrows used to construct the colimit.

The second proof defines an isomorphism between $L\,A \to B$ and $A \to R\,B$ for some functor R and for all A and B. This isomorphism is said to be an *adjunction between* L *and* R, and R is said to be a *right adjoint of* L. Bird and Paterson prove that if L has a right adjoint then the equation has a unique solution. Taking $A{=}T$ and defining the hofunctor L by $L\,X{=}X \cdot M$ and $L\,\beta{=}\beta_M$, they construct a right adjoint to L and thereby define an isomorphism between generalised folds and higher-order simple folds. The case of efficient folds has not been studied, however so we must rely on the first proof to prove their universal property.

3.6 Map-fusion law for efficient folds

Observe that since T is a functor we have

$$(\!|f \mid g|\!)_F \cdot T\,h \cdot T\,k \;=\; (\!|f \mid g|\!)_F \cdot T\,(h \cdot k)$$

From the specification of efficient folds we can immediately conclude the following simple map-fusion law.

$$\{\![f \mid g \mid h]\!\}_F \cdot T\,k \;=\; \{\![f \mid g \mid h \cdot k]\!\}_F$$

This law has no conditions, unlike its counterpart for generalised folds, and in this respect it resembles the map-fusion law for standard folds. However, as generalised folds are efficient folds, we would expect the law for generalised folds to be a special case of the law for efficient folds. Consequently, we shall try to derive a more general law for efficient folds. For a start, by rewriting efficient folds to generalised folds composed with maps, we can use the map-fusion law for generalised folds to show

$$\begin{aligned} \{\![f \mid g \mid h]\!\}_{NestF} \cdot Nest\,k \;&=\; \{\![f \cdot Base\;(h \cdot k, id) \mid g' \mid id]\!\}_{NestF} \\ &\Leftarrow\; g \cdot Pair\;(h \cdot k) \;=\; h \cdot k \cdot g' \end{aligned}$$

Taking $h{=}id$ recovers the law for generalised folds. Alternatively, we could try writing the map as an efficient fold and using the fold-fusion law to derive some conditions.

$$\{\![f \mid g \mid h]\!\}_{NestF} \cdot \{\![\alpha \mid id \mid k]\!\}_{NestF} \quad = \quad \{\![f' \mid g' \mid h']\!\}_{NestF}$$

However, the fold-fusion law for efficient reductions cannot be used here because the map is not an efficient reduction. The derivation of the fold-fusion law for efficient folds can be redone using more general fold-fusion laws for generalised folds. However, apart from the special case where $M' = Id$, which requires $g' = id$, these laws all introduce conditions that are difficult to satisfy.

Finally, we could try to derive a map-fusion law direct from the universal property of efficient folds, but unfortunately, it is difficult to find values of the mapping χ that give satisfiable conditions.

In conclusion, none of these three ideas produce an interesting generalisation of the map-fusion laws of generalised folds and efficient folds.

Chapter 4

Other folds for non-linear datatypes

In the previous chapter, we defined generalised and efficient fold operators for linear nested datatypes. Now we shall extend these operators to non-linear datatypes. More precisely, we defined the fold operators for a functor T of the form below, where F_1 had to be a constant hofunctor, that is, a hofunctor that always returns the same functor.

$$\begin{array}{rcl} T & \approx & F\,T \\ F\,X & = & B \cdot \langle \mathit{Id}, X \cdot F_1\,X \rangle \end{array}$$

Although this form gives only some linear datatypes, the definitions could easily be generalised. In this chapter, we remove the restriction on F_1. We also give generic versions of the fold operators, which we can use later in this thesis to define and reason about generic operations.

Our running example of a non-linear datatype will be the datatype of bushes given by taking $B = \mathit{Base}$ and $F_1\,X = X$.

$$\begin{array}{rcl} \mathit{Bush} & \approx & \mathit{BushF}\;\mathit{Bush} \\ \mathit{BushF}\,X & = & \mathit{Base} \cdot \langle \mathit{Id}, X \cdot \mathit{BushF}_1\,X \rangle \\ \mathit{BushF}_1\,X & = & X \end{array}$$

In Haskell, this is written as

$$\textbf{data}\ \mathit{Bush}\ a \;=\; \mathit{NilB} \mid \mathit{ConsB}\ (a, \mathit{Bush}\ (\mathit{Bush}\ a))$$

Note that the recursive use of *Bush* is modified by *Bush* itself and this is what makes *Bush* non-linear. A bush is thereby defined as a list consisting of an element followed by a bush followed by a bush of bushes and so on. The variant of *Bush* obtained by removing the *NilB* case is used in [Hin00b] to implement trie-structures (basically efficient finite maps) that are indexed by unlabelled binary trees. Another non-linear datatype, given in [BP99a], improves on the datatype of de Bruijn terms presented in the previous chapter by showing, for the sake of efficiency, the sharing of lambda-expressions between lambda-terms.

Before continuing, we note that the generalised fold operator for bushes must be different in form from the operator for nests. The reason for this is that the map operator must be used with the type

$$\mathit{BushF}_1\, \mathit{Bush} \cdot M \stackrel{\cdot}{\to} M \cdot \mathit{BushF}_1\, R$$

because the map operator for nests is used with the type

$$\mathit{NestF}_1\, \mathit{Nest} \cdot M \stackrel{\cdot}{\to} M \cdot \mathit{NestF}_1\, R$$

and the first of these is not independent of *Bush* so it cannot be a parameter to the generalised fold operator for bushes. Here, NestF_1 is defined by writing *Nest* in the form

$$\begin{aligned} \mathit{Nest} &\approx \mathit{NestF}\, \mathit{Nest} \\ \mathit{NestF}\, X &= \mathit{Base} \cdot \langle \mathit{Id}, X \cdot \mathit{NestF}_1\, X \rangle \\ \mathit{NestF}_1\, X &= \mathit{Pair} \end{aligned}$$

The layout of this chapter is as follows. Section 4.1 defines a generalised fold operator for bushes. Section 4.2 describes an alternative, though equally expressive, grammar for polynomial hofunctors. Section 4.3 uses induction on the structure of this grammar to define a generic generalised fold operator. Section 4.4 explains how we can use this operator to define generic operations and illustrates this by defining a generic sum operation. Sections 4.5 and 4.6 adapt Bird and Paterson's generic map-fusion and fold-fusion laws to make them more useful for calculation. Finally, Section 4.7 derives a generic efficient fold operator.

4.1 Generalised folds for bushes

Suppose we define the generalised fold operator for bushes along the same lines as the corresponding operator for nests, below.

$$(\!|f \mid g|\!)_{NestF} \cdot \alpha_{NestF} \;=\; f \cdot Base\ (id, (\!|f \mid g|\!)_{NestF} \cdot Nest\ g)$$

As we have just explained, the map operator for bushes must be used with some argument

$$rest_1 \;:\; BushF_1\, Bush \cdot M \dot{\to} M \cdot BushF_1\, R$$

Although $rest_1$ is dependent on $Bush$ and cannot be a parameter to the fold operator, we can construct it from a parameter $rest_2$ that is not dependent on $Bush$.

$$rest_2 \;:\; BushF_2\, Bush \cdot M \dot{\to} M \cdot BushF_2\, R$$

Here $BushF_2$ is defined by putting $BushF_1$ into the form

$$\begin{aligned} BushF_1\, X &= Outr \cdot \langle Id, X \cdot BushF_2\, X\rangle \\ BushF_2\, X &= Id \end{aligned}$$

Observe that the right-hand side of the definition of $(\!|-|-|\!)_{NestF}$ above has source type $NestF\ Nest \cdot M$ and is constructed from a parameter g of source type $NestF_1\, Nest \cdot M$ and another parameter f. In the same way, we can construct $rest_1$ from $rest_2$ and a new parameter g_1. The generalised fold operator for bushes that we end up with is

$$\begin{aligned} (\!|-|-|-|\!)_{BushF} \;:\; & (Base \cdot \langle M, R \cdot Outr \cdot \langle Id, R\rangle\rangle \dot{\to} R) \to \\ & (Outr \cdot \langle M, R\rangle \dot{\to} M \cdot Outr \cdot \langle Id, R\rangle) \to \\ & (Id \cdot M \dot{\to} M \cdot Id) \to \\ & Bush \cdot M \dot{\to} R \end{aligned}$$

$$(\!|f \mid g_1 \mid g_2|\!)_{Bush} \cdot \alpha = f \cdot Base\ (id,\ (\!|f \mid g_1 \mid g_2|\!)_{Bush} \cdot Bush\ rest_1)$$

$$\begin{aligned} rest_1 &: Outr \cdot \langle M, Bush \cdot M\rangle \dot{\to} M \cdot Outr \cdot \langle Id, R\rangle \\ rest_1 &= g_1 \cdot Outr\ (id,\ (\!|f \mid g_1 \mid g_2|\!)_{Bush} \cdot Bush\ rest_2) \end{aligned}$$

$$\begin{aligned} rest_2 &: Id \cdot M \dot{\to} M \cdot Id \\ rest_2 &= g_2 \end{aligned}$$

Note that this operator has three arguments (written between two vertical bars) the first of which is the join of two functions. When we implement the generalised fold operator in Haskell, we assume for the sake of brevity that the third parameter, which has the type of an identity function, always *is* an identity function and can therefore be omitted. In other words, we have that $(\![nil, cons \mid more \mid id]\!)_{BushF}$ is implemented by *gfoldbush nil cons more* where *gfoldbush* is defined by

$$
\begin{aligned}
gfoldbush \;::\; & \textbf{forall}\; a\; m\; r. \\
& (\textbf{forall}\; a.\; r\; a) \rightarrow \\
& (\textbf{forall}\; a.\; (m\; a, r\; (r\; a)) \rightarrow r\; a) \rightarrow \\
& (\textbf{forall}\; a.\; r\; a \rightarrow m\; (r\; a)) \rightarrow \\
& Bush\; (m\; a) \rightarrow r\; a
\end{aligned}
$$

$$gfoldbush\; nil\; cons\; more\; NilB = nil$$

$$
\begin{aligned}
& gfoldbush\; nil\; cons\; more\; (ConsB\; (a, x)) = cons\; (a, \\
& \quad gfoldbush\; nil\; cons\; more\; (bush\; (more \cdot gfoldbush\; nil\; cons\; more)\; x))
\end{aligned}
$$

$$
\begin{array}{lcl}
bush & :: & (a \rightarrow b) \rightarrow Bush\; a \rightarrow Bush\; b \\
bush\; f\; NilB & = & NilB \\
bush\; f\; (ConsB\; (a, x)) & = & ConsB\; (f\; a, bush\; (bush\; f)\; x)
\end{array}
$$

Let *gf* abbreviate *gfoldbush nil cons more*. The type assertions below help us to understand how *gfoldbush* works.

$$
\begin{array}{rcl}
x & : & Bush\; (Bush\; (m\; a)) \\
bush\; gf\; x & : & Bush\; (r\; a) \\
bush\; (more \cdot gf)\; x & : & Bush\; (m\; (r\; a)) \\
gf\; (bush\; (more \cdot gf)\; x) & : & r\; (r\; a)
\end{array}
$$

The fold turns bushes (starting from the innermost in the subcall) into r's but it can only be applied to bushes of m's, so before replacing the outer bush in the type of x, the fold applies the function *more* to introduce the type constructor m.

Note, however, that the generalised fold for bushes given in [Hin99a] is very different from ours. Although the folds are equally powerful, the parameters

are different in type and number and they also have a different role. A more detailed comparison is given at the end of [Hin99a] itself.

It is also interesting to compare the assertions above with the corresponding assertions for *gfoldnest*.

$$\begin{array}{rcl} x & : & \mathit{Nest}\ (\mathit{Pair}\ (m\ a)) \\ \mathit{nest}\ \mathit{bin}\ x & : & \mathit{Nest}\ (m\ (\mathit{Pair}\ a)) \\ \mathit{gfoldnest}\ \mathit{nil}\ \mathit{cons}\ \mathit{bin}\ (\mathit{nest}\ \mathit{bin}\ x) & : & r\ (\mathit{Pair}\ a) \end{array}$$

Here, the parameter *bin* does not reintroduce the functor m, but merely commutes it so that the fold can be applied recursively.

4.2 Alternative grammar for polynomial hofunctors

The generalised fold operators for nests and bushes are different in form and in the number of parameters, so clearly both of these characteristics are determined by the datatype. We shall therefore define the generic operator by induction on the structure of polynomial hofunctors. The grammar we gave in Chapter 2 for polynomial hofunctors has separate cases for constant functors, identity functors, products, coproducts and composition. The grammars given in [MG01] and [Hin99a] have a similar number of cases, though they are not exactly the same, and the definitions there of generic operators are given by case analysis. This means that any proofs that use the definitions must also proceed by case analysis, which makes them longer and more difficult to understand. Since we have a lot of proofs in this thesis, we shall follow [BP99b] and define the generic generalised fold operator using a grammar for polynomial hofunctors that has only one case. Every polynomial hofunctor can and will be put into the form

$$F\,X \quad = \quad B \cdot \langle \mathit{Id}, X \cdot F_1\,X, \ldots, X \cdot F_n\,X \rangle$$

The F_i are expressed in a similar form and B is an $(n{+}1)$-ary functor.

4.3 Generic generalised fold operator

Now we use this datatype to define the generic generalised fold operator. The operator must return a fold of type $T \cdot M \dot{\to} R$ when given a *main parameter* f that contains replacements for the constructor functions.

$$f \; : \; B \cdot \langle M, R \cdot F_1 \, R \rangle \dot{\to} R$$

The operator for nests has one auxiliary parameter and the operator for bushes has two auxiliary parameters. The generic operator therefore requires, as an argument, an indexed collection of parameters γ. We associate with each subsidiary hofunctor F_i, an *auxiliary parameter*

$$\gamma_i \; : \; B_i \cdot \langle M, R \cdot F_j \, R \rangle \dot{\to} M \cdot F_i \, R$$

If $(\!| f \parallel \gamma |\!)_F$ is a reduction of type $T\ b \to c$ then f has type $B\ (b, c) \to c$ and γ_i has type $B_i\ (b, c) \to b$. Note that the generic operator is written with a double bar instead of a single bar, which is used for specific datatypes.

The universal property for the generic generalised fold operator is as follows.

$$\begin{aligned} (\!| f \parallel \gamma |\!)_F \quad &: \quad T \cdot M \dot{\to} R \\ h = (\!| f \parallel \gamma |\!)_F \quad &\equiv \quad h \cdot \alpha \; = \; f \cdot B\ (id,\ h \cdot\ T\ (\Phi_1\ (\gamma,\ h))) \end{aligned}$$

Here we associate a term $\Phi_i\ (\gamma, h)$ with each i, generalising the functions $rest_i$, which in the case of bushes are defined only for $i \in \{1, 2\}$.

$$\begin{aligned} \Phi_i\ (\gamma,\ h) \quad &: \quad F_i\ T \cdot M \dot{\to} M \cdot F_i\ R \\ \Phi_i\ (\gamma,\ h) \quad &= \quad \gamma_i \cdot B_i\ (id,\ h \cdot\ T\ (\Phi_j\ (\gamma,\ h))) \end{aligned}$$

Note that "with each i" above is shorthand for "for all i such that F_i is a subsidiary hofunctor of F". The generic simple fold operator can now be expressed in terms of the generic generalised fold operator as follows.

$$\begin{aligned} (\!| - |\!)_F \quad &: \quad (F\ R \dot{\to} R) \to (T \dot{\to} R) \\ (\!| f |\!)_F \quad &= \quad (\!| f \parallel ident |\!)_F \end{aligned}$$

Here, $ident$ is defined by $ident_i = id_{F_i\ R}$.

4.4 Generic summation

The generic generalised fold operator enables us to write generic operations on nested datatypes as generalised folds. To illustrate this, we shall now define a generic summation operation *sum* such that sum_T has type $T\ Int \to Int$ and takes a T-structure of integers to the sum of its elements. Often we shall generalise from regular functors but here we shall generalise from nests. We know that $sum_{Nest} = sumnest$ so

$$\begin{array}{lcl} sumnest & : & Nest\ Int \to Int \\ sumnest & = & (\![kzero, plus \mid plus]\!)_{NestF} \end{array}$$

The parameters to this fold have types $Base\ (Int, Int) \to Int$ and $Pair\ Int \to Int$, so we can rewrite *sumnest* as follows.

$$\begin{array}{lcl} sumnest & : & Nest\ Int \to Int \\ sumnest & = & (\![sum_{Base} \mid sum_{Pair}]\!)_{NestF} \end{array}$$

Now we generalise. Suppose we define *gsum* by $gsum_i = sum_{B_i}$, for all i. Then we have

$$\begin{array}{lcl} sum_T & : & T\ Int \to Int \\ sum_T & = & (\![sum_B \parallel gsum]\!)_F \end{array}$$

We call this a nested fixpoint case. It is defined in terms of polynomial cases that we shall soon supply. These will be chosen from type considerations, examining unary and binary functors separately. Although we have motivated the case for nests (and hence the nested fixpoint case) from type considerations, we will often motivate it instead from a known regular fixpoint case. Now for an example.

$$\begin{array}{lcl} sum_{Bush} & : & Bush\ Int \to Int \\ sum_{Bush} & = & (\![sum_{Base} \mid sum_{Outr} \mid sum_{Id}]\!)_{BushF} \end{array}$$

Further polynomial cases are required to define sum_{Base} and sum_{Outr} and sum_{Id}. We construct these polynomial cases by writing out their types pointwise. We start with the unary functors.

$$\begin{array}{lcl} sum_{K_A} & : & a \to Int \\ sum_{Id} & : & Int \to Int \\ sum_{F+G} & : & F\ Int\ +\ G\ Int \to Int \\ sum_{F\times G} & : & F\ Int\ \times\ G\ Int \to Int \\ sum_{F\cdot G} & : & F\ (G\ Int) \to Int \end{array}$$

Now we choose the values.

$$\begin{array}{lcl}
sum_{K_A} & = & kzero \\
sum_{Id} & = & id_{Int} \\
sum_{F+G} & = & [sum_F, sum_G] \\
sum_{F \times G} & = & plus \cdot (sum_F \times sum_G) \\
sum_{F \cdot G} & = & sum_F \cdot F\ sum_G
\end{array}$$

Now we do the same for the binary functors.

$$\begin{array}{lcl}
sum_{KK_A} & : & a \to Int \\
sum_{Outl} & : & Int \to Int \\
sum_{Outr} & : & Int \to Int \\
sum_{F \cdot \langle G,H \rangle} & : & F\ (G\ Int, H\ Int) \to Int
\end{array}$$

The values are

$$\begin{array}{lcl}
sum_{KK_A} & = & kzero \\
sum_{Outl} & = & id_{Int} \\
sum_{Outr} & = & id_{Int} \\
sum_{F \cdot \langle G,H \rangle} & = & sum_F \cdot F\ (sum_G, sum_H)
\end{array}$$

We can easily generalise these cases to n-ary functors, but we shall limit ourselves to unary and binary functors in this thesis. Let us check the cases above by evaluating sum_{Base}

$$\begin{array}{ll}
 & sum_{Base} \\
= & sum_{KK_1 + Outl \times Outr} \\
= & [sum_{KK_1}, plus \cdot (sum_{Outl} \times sum_{Outr})] \\
= & [kzero, plus \cdot (id_{Int} \times id_{Int})] \\
= & [kzero, plus]
\end{array}$$

We derive a case for sum_+ from the case for $sum_{F+G} = sum_{+ \cdot \langle F,G \rangle}$.

$$\begin{array}{ll}
 & sum_+ \cdot (sum_F + sum_G) \\
= & sum_{+ \cdot \langle F,G \rangle} \\
= & sum_{F+G} \\
= & [sum_F, sum_G] \\
= & [id, id] \cdot (sum_F + sum_G)
\end{array}$$

So we conclude

$$sum_+ \quad = \quad [id, id]$$

For products, a similar calculation gives

$$sum_{\times} \quad = \quad plus$$

We shall often need to derive cases for the bifunctors + and × in this way; sometimes we derive the case for bifunctor composition at the same time. Although we use type considerations to motivate the polynomial and fixpoint cases, the fold-equivalence law of Chapter 8 will allow us to deduce the nested fixpoint case from the composition case and the regular fixpoint case alone.

The generic definition of *sum* should be contrasted with the definition in [Hin99b], where the fixpoint case is omitted but inferred automatically. Using our notation, Hinze would calculate sum_{Nest} as follows.

$$\begin{aligned} & sum_{Nest} \\ = \; & sum_{K_1 + Id \times Nest \cdot Pair} \\ = \; & [sum_{K_1}, plus \cdot (sum_{Id} \times sum_{Nest} \cdot Nest\ sum_{Pair})] \\ = \; & [kzero, plus \cdot (id \times (sum_{Nest} \cdot Nest\, plus))] \end{aligned}$$

In [Hin00c], Hinze argues further that the composition case can also be omitted and then inferred from the remaining cases when the generic operation is specialised.

With the polynomial cases defined we can now implement sum_{Bush} in Haskell; recall that *gfoldbush nil cons more* implements $(\![nil, cons \mid more \mid id]\!)_{BushF}$.

$$\begin{aligned} sumbush \;&::\; Bush\ Int \to Int \\ sumbush \;&=\; unKInt \cdot gfoldbush\ zero'\, plus'\, id \cdot bush\, KInt \end{aligned}$$

$$\begin{aligned} zero' \;&:\; KInt\ a \\ zero' \;&=\; KInt\, 0 \end{aligned}$$

$$\begin{aligned} plus' \;&:\; (KInt\ a, KInt\ a) \to KInt\ a \\ plus'\ (KInt\, x, KInt\, y) \;&=\; KInt\ (x + y) \end{aligned}$$

4.5 Generic map-fusion law for generalised folds

In [BP99b], Bird and Paterson use the universal property of the generic generalised fold operator to derive a generic map-fusion law. We presented the

derivation for the special case of nests, in the previous chapter.

Generic map-fusion law If we have

$$\Phi_1\ (\gamma,\ (\!|f \parallel \gamma|\!)_F) \cdot F_1\ T\ k \quad = \quad k_{F_1\ R} \cdot \Phi_1\ (\gamma',\ (\!|f \parallel \gamma|\!)_F \cdot T\ k)$$

then we can conclude

$$(\!|f \parallel \gamma|\!)_F \cdot T\ k \quad = \quad (\!|f \cdot B\ (k, id) \parallel \gamma'|\!)_F$$

Here, $(\!|f \parallel \gamma|\!)_F$ has type $T \cdot M \dot{\to} R$ and k has type $M' \dot{\to} M$. Here is the law instantiated to bushes, after dropping the subscript on k for concision.

Map-fusion law for bushes Given the typings

$$\begin{aligned}
f &: Base \cdot \langle M, R \cdot R\rangle \dot{\to} R \\
g_1 &: R \dot{\to} M \cdot R \\
g_2 &: M \dot{\to} M \\
g_1' &: R \dot{\to} M' \cdot R \\
g_2' &: M' \dot{\to} M' \\
k &: M' \dot{\to} M
\end{aligned}$$

we have

$$(\!|f \mid g_1 \mid g_2|\!)_{BushF} \cdot Bush\ k = (\!|f \cdot Base\ (k, id) \mid g_1' \mid g_2'|\!)_{BushF}$$

$$\begin{aligned}
\Leftarrow \quad & g_1 \cdot Outr\ (k, (\!|f \parallel \gamma|\!)_F \cdot Bush\ (g_2 \cdot Id\ k)) \\
& \quad = k \cdot g_1' \cdot Outr\ (id, (\!|f \parallel \gamma|\!)_F \cdot Bush\ (k \cdot g_2'))
\end{aligned}$$

A necessary condition for this condition is the conjunction

$$g_1 \cdot Outr\ (k, id) \ = \ k \cdot g_1' \qquad \text{and} \qquad g_2 \cdot Id\ k \ = \ k \cdot g_2'$$

Neither of these clauses have fold or map operations so they are easier to verify. Because the functors *Outr* and *Id* have been made explicit, we can spot patterns in common with the law for nests. Comparing these conditions with the condition for nests, we can propose a generic map-fusion law with conditions that are free of maps and folds.

Improved generic map-fusion law If we have for all i,

$$\gamma_i \cdot B_i\ (k,\ id) \quad = \quad k_{\ F_1\ R} \cdot \gamma'_i$$

then we conclude

$$(\![f \parallel \gamma]\!)_F \cdot T\ k \quad = \quad (\![f \cdot B\ (k, id) \parallel \gamma']\!)_F$$

Now we show that the collection of conditions above implies the single condition of the generic law derived in [BP99b]. We reason by induction on the structure of hofunctors. Suppose that F_i is defined by

$$F_i\ X \quad = \quad B_i \cdot \langle Id, X \cdot F_j\ X \rangle$$

Our induction hypothesis is

$$\Phi_j\ (\gamma, (\![f \parallel \gamma]\!)_F) \cdot F_j\ T\ k \quad = \quad k \cdot \Phi_j\ (\gamma', (\![f \parallel \gamma]\!)_F \cdot T\ k)$$

Now we argue:

$$\begin{array}{ll}
& \Phi_i\ (\gamma,\ (\![f \parallel \gamma]\!)_F) \cdot F_i\ T\ k \\
= & \quad \{\ \text{definition of } \Phi \text{ and } F_i\ \} \\
& \gamma_i \cdot B_i\ (id\ , (\![f \parallel \gamma]\!)_F \cdot T\ (\Phi_j\ (\gamma, (\![f \parallel \gamma]\!)_F))) \cdot B_i\ (k,\ T\ (F_j\ \ T\ \ k)) \\
= & \quad \{\ B_i \text{ and } T \text{ are functors}\ \} \\
& \gamma_i \cdot B_i\ (k,\ id) \cdot B_i\ (id, (\![f \parallel \gamma]\!)_F \cdot T\ (\Phi_j\ (\gamma, (\![f \parallel \gamma]\!)_F) \cdot F_j\ T\ k)) \\
= & \quad \{\ \textbf{assume}\ \gamma_i \cdot B_i\ (k,\ id) = k \cdot \gamma'_i\ \} \\
& k \cdot \gamma'_i \cdot B_i\ (id, (\![f \parallel \gamma]\!)_F \cdot T\ (\Phi_j\ (\gamma', (\![f \parallel \gamma]\!)_F) \cdot F_j\ T\ k)) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& k \cdot \gamma'_i \cdot B_i\ (id, (\![f \parallel \gamma]\!)_F \cdot T\ (k \cdot \Phi_j\ (\gamma', (\![f \parallel \gamma]\!)_F \cdot T\ k))) \\
= & \quad \{\ \text{definition of } \Phi;\ T \text{ is a functor}\ \} \\
& k \cdot \Phi_i\ (\gamma', (\![f \parallel \gamma]\!)_F \cdot T\ k)
\end{array}$$

4.6 Generic fold-fusion law for generalised folds

The generic fold-fusion law derived in [BP99b] is

Generic fold-fusion law If there exists a function $p : F_1\, R \cdot M' \dot{\to} M' \cdot F_1\, R'$ such that

$$k \cdot f = f' \cdot B\ (id, k \cdot R\, p)$$

and

$$M\ p \cdot \Phi_1\ (\gamma,\ (\![f \parallel \gamma]\!)_F) = \Phi_1\ (\gamma',\ k \cdot (\![f \parallel \gamma]\!)_F)$$

then we can conclude

$$k \cdot ((\![f \parallel \gamma]\!)_F)_{M'} = (\![f' \parallel \gamma']\!)_F$$

Here $(\![f \parallel \gamma]\!)_F$ has type $T \cdot M \dot{\to} R$ and k has type $R \cdot M' \dot{\to} R'$. Instantiated to bushes the law is

Fold-fusion law for bushes Given the typings

$$\begin{array}{rcl} f & : & Base \cdot \langle M, R \cdot R\rangle \dot{\to} R \\ f' & : & Base \cdot \langle M \cdot M', R' \cdot R'\rangle \dot{\to} R' \\ g_1 & : & R \dot{\to} M \cdot R \\ g_1' & : & R' \dot{\to} M \cdot M' \cdot R' \\ g_2 & : & M \dot{\to} M \\ g_2' & : & M \cdot M' \dot{\to} M \cdot M' \\ k & : & R \cdot M' \dot{\to} R' \end{array}$$

if we have

$$k \cdot f\ = f' \cdot Base\ (id, k \cdot R\ p)$$

and we also have

$$\begin{array}{l} M\ p \cdot g_1 \cdot Outr\ (id, (\![f \mid g_1 \mid g_2]\!)_{BushF} \cdot Bush\ g_2) \\ \qquad = g_1' \cdot Outr\ (id, k \cdot (\![f \mid g_1 \mid g_2]\!)_{BushF} \cdot Bush\ g_2') \end{array}$$

with $p : R \cdot M' \dot{\to} M' \cdot R'$ then we have

$$k \cdot (\![f \mid g_1 \mid g_2]\!)_{BushF} \quad = \quad (\![f' \mid g_1' \mid g_2']\!)_{BushF}$$

Now we wish to find as sufficient conditions some rewriting equations without fold or map operations that together turn the left-hand side of the second

condition above into the right-hand side. We guess by analogy with the law for nests that one of the rewrites is, for some $q : M' \dot{\to} M'$,

$$M\ q \cdot g_2 \quad = \quad g_2'$$

This can be preceeded by a rewrite that uses the naturality of generalised folds to introduce the $M\ q$ term. However, this rewrite does not need to be a condition as it is an immediate consequence of the naturality of the fold.

$$R\ q \cdot (\!| f \mid g_1 \mid g_2 |\!)_{BushF} \quad = \quad (\!| f \mid g_1 \mid g_2 |\!)_{BushF} \cdot Bush\ (M\ q)$$

Finally, the term $R\ q$ can be introduced by the rewrite

$$M\ p \cdot g_1 \quad = \quad g_1' \cdot Outr\ (id, k \cdot R\ q)$$

Comparing these equations with the second condition in the law for nests suggests the following generic law, the conditions of which feature neither folds nor maps and are therefore easy to verify for individual cases.

Improved generic fold-fusion law If we have an indexed collection of functions p, such that for all i with $F_i\ X \ = \ B_i \cdot \langle Id, X \cdot F_j\ X \rangle$ the function p_i has type $F_i\ R \cdot M' \dot{\to} M \cdot F_i\ R'$

$$M\ p_i \cdot \gamma_i \quad = \quad \gamma_i' \cdot B_i\ (id, k \cdot R\ p_j)$$

and

$$k \cdot f \quad = \quad f' \cdot B\ (id, k \cdot R\ p_1)$$

then we can conclude that

$$k \cdot (\!| f \parallel \gamma |\!)_F \quad = \quad (\!| f' \parallel \gamma' |\!)_F$$

We use structural induction to show that the indexed collection of conditions in the rule above implies the second condition of Bird and Paterson's law. Our induction hypothesis is

$$M\ p_j \cdot \Phi_j\ (\gamma,\ (\!| f \parallel \gamma |\!)_F) \quad = \quad \Phi_j\ (\gamma',\ k \cdot (\!| f \parallel \gamma |\!)_F)$$

We reason as follows:

$$\begin{array}{ll} & M\ p_i \cdot \Phi_i\ (\gamma,\ (\!| f \parallel \gamma |\!)_F) \\ = & \{ \text{ definition of } \Phi \ \} \\ & M\ p_i \cdot \gamma_i \cdot B_i\ (id, (\!| f \parallel \gamma |\!)_F \cdot T\ (\Phi_j\ (\gamma,\ (\!| f \parallel \gamma |\!)_F))) \end{array}$$

$$
\begin{array}{ll}
= & \{ \; B_i \text{ is a functor; } \textbf{assume } M\, p_i \cdot \gamma_i = \gamma'_i \cdot B_i \; (id, k \cdot R\, p_j) \; \} \\
& \gamma'_i \cdot B_i \; (id, k \cdot R\, p_j \cdot (\!| f \parallel \gamma |\!)_F \cdot T \; (\Phi_j \; (\gamma, \, (\!| f \parallel \gamma |\!)_F))) \\
= & \{ \; \text{naturality of } (\!| f \parallel \gamma |\!)_F \text{ ; } T \text{ is a functor} \; \} \\
& \gamma'_i \cdot B_i \; (id, k \cdot (\!| f \parallel \gamma |\!)_F \cdot T \; (M\, p_j \cdot \Phi_j \, (\gamma, \, (\!| f \parallel \gamma |\!)_F))) \\
= & \{ \; \text{induction hypothesis} \; \} \\
& \gamma'_i \cdot B_i \; (id, k \cdot (\!| f \parallel \gamma |\!)_F \cdot T \; (\Phi_j \; (\gamma', \, k \cdot (\!| f \parallel \gamma |\!)_F))) \\
= & \{ \; \text{definition of } \Phi \; \} \\
& \Phi_i \; (\gamma', \, k \cdot (\!| f \parallel \gamma |\!)_F)
\end{array}
$$

This law has a simpler form for reductions.

Fold-fusion law for reductions If we have for all i,

$$\gamma_i \;\; = \;\; \gamma'_i \cdot B_i \; (id, k)$$

and in addition

$$k \cdot f \;\; = \;\; f' \cdot B \; (id, k)$$

then we can conclude

$$k \cdot (\!| f \parallel \gamma |\!)_F \;\; = \;\; (\!| f' \parallel \gamma' |\!)_F$$

4.7 Efficient folds for non-linear datatypes

The generic efficient fold operator is specified by, for T with $T \approx F\ T$,

$$\{\![f \parallel \gamma \parallel h]\!\}_F \;\; = \;\; (\!| f \parallel \gamma |\!)_F \cdot T\, h$$

Note that we use double vertical bars to separate arguments in the generic version of the operator, just as we did with generalised folds. Now we derive a direct definition of the operator, repeating the steps from the special case of nests.

$$
\begin{array}{ll}
& \{\![f \parallel \gamma \parallel h]\!\}_F \cdot \alpha \\
= & \{ \; \text{specification of efficient fold} \; \} \\
& (\!| f \parallel \gamma |\!)_F \cdot T \; h \cdot \alpha
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ naturality of } \alpha \ \} \\
& (\!|f \parallel \gamma|\!)_F \cdot \alpha \cdot B\ (h, T\ (F_1\ T\ h)) \\
= & \{ \text{ definition of generalised fold } \} \\
& f \cdot B\ (id, (\!|f \parallel \gamma|\!)_F \cdot T\ (\Phi_1\ (\gamma, (\!|f \parallel \gamma|\!)_F))) \cdot B\ (h, T\ (F_1\ T\ h)) \\
= & \{ \ B \text{ and } T \text{ are functors } \} \\
& f \cdot B\ (h, (\!|f \parallel \gamma|\!)_F \cdot T\ (\Phi_1\ (\gamma, (\!|f \parallel \gamma|\!)_F) \cdot F_1\ T\ h)) \\
= & \{ \text{ specification of efficient fold } \} \\
& f \cdot B\ (h, \{\![f \parallel \gamma \parallel \Phi_1\ (\gamma, (\!|f \parallel \gamma|\!)_F) \cdot F_1\ T\ h]\!\}_F)
\end{array}
$$

If T is non-linear then $F_1\ T$ is not a constant functor and there are wasteful maps inside both $F_1\ T\ h$ and the term $\Phi_1\ (\gamma, (\!|f \parallel \gamma|\!)_F)$. Once again, our solution is to specify a more general operator Φ' by introducing an accumulating parameter.

$$
\Phi'_i\ (\gamma, (\!|f \parallel \gamma|\!)_F, h) \quad = \quad \Phi_i\ (\gamma, (\!|f \parallel \gamma|\!)_F) \cdot F_i\ T\ h
$$

We have also generalised F_1 to a hofunctor F_i, which we define for some F_j and B_i by

$$
F_i\ X \quad = \quad B_i \cdot \langle Id, X \cdot F_j\ X \rangle
$$

Now we derive a direct recursive definition of Φ'.

$$
\begin{array}{ll}
& \Phi'_i\ (\gamma, (\!|f \parallel \gamma|\!)_F, h) \\
= & \{ \text{ specification of } \Phi' \ \} \\
& \Phi_i\ (\gamma, (\!|f \parallel \gamma|\!)_F) \cdot F_i\ T\ h \\
= & \{ \text{ definition of } \Phi \text{ and } F_i \ \} \\
& \gamma_i \cdot B_i\ (id, (\!|f \parallel \gamma|\!)_F \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F))) \cdot B_i\ (h, T\ (F_j\ T\ h)) \\
= & \{ \ B_i \text{ and } T \text{ are functors } \} \\
& \gamma_i \cdot B_i\ (h, (\!|f \parallel \gamma|\!)_F \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F) \cdot F_j\ T\ h)) \\
= & \{ \text{ specification of efficient fold and } \Phi' \ \} \\
& \gamma_i \cdot B_i\ (h, \{\![f \parallel \gamma \parallel \Phi'_j\ (\gamma, (\!|f \parallel \gamma|\!)_F, h)_F]\!\})
\end{array}
$$

The universal property of the efficient fold operator is therefore given by

$$
\begin{array}{rcl}
\{\![f \parallel \gamma \parallel h]\!\}_F & : & T \cdot M' \cdot K_{a'} \stackrel{\cdot}{\to} R \cdot K_a \\
\chi\ h = \{\![f \parallel \gamma \parallel h]\!\}_F & \equiv & \chi\ h \cdot \alpha = f \cdot B\ (h, \chi\ (\xi_1\ h))
\end{array}
$$

$$\xi_i \; h \quad : \quad F_i \, T \cdot M' \cdot K_{a'} \dot{\rightarrow} M \cdot F_i \, R \cdot K_a$$
$$\xi_i \; h \quad = \quad \gamma_i \cdot B_i \; (h, \chi \; (\xi_j \, h))$$

The definition of ξ is implicitly for all i, where i and j are related by $F_i \, X = B_i \cdot \langle Id, X \cdot F_j \, X \rangle$. As an example, the efficient fold operator for bushes is given by taking $B = Base$ and $F_1 \, X = X$. In Haskell it is written as follows.

$$\begin{aligned}
\mathit{efoldbush} \;::\; & \textbf{forall}\; a\; a'\; m\; m'\; r. \\
& (\textbf{forall}\; b.\; r\; b) \rightarrow \\
& (\textbf{forall}\; b.\; (m\; b, r\; (r\; b)) \rightarrow r\; b) \rightarrow \\
& (\textbf{forall}\; b.\; r\; b \rightarrow m\; (r\; b)) \rightarrow \\
& (m'\; a' \rightarrow m\; a) \rightarrow \\
& \mathit{Bush}\; (m'\; a') \rightarrow r\; a
\end{aligned}$$

$$\begin{aligned}
& \mathit{efoldbush\; nil\; cons\; more\; one\; Nil} \;=\; \mathit{nil} \\
& \mathit{efoldbush\; nil\; cons\; more\; one}\; (\mathit{Cons}\; (a, x)) \\
& \qquad = \; \mathit{cons}\; (\mathit{one}\; a, \mathit{ef}\; (\mathit{more} \cdot \mathit{ef}\; \mathit{one})\; x) \\
& \qquad\qquad \textbf{where}\; \mathit{ef} = \mathit{efoldbush\; nil\; cons\; more}
\end{aligned}$$

We have omitted the parameter that has the type of an identity function, just like we did when writing *gfoldbush*

4.8 Map-fusion law for efficient folds

Finally, the map-fusion law for efficient folds on linear datatypes generalises to arbitrary datatypes.

$$\{\![f \parallel \gamma \parallel h]\!\}_F \cdot T \, k \quad = \quad \{\![f \parallel \gamma \parallel h \cdot k]\!\}_F$$

We shall derive the corresponding fold-fusion law, and examine some of its special cases, at the end of the next chapter.

Chapter 5

Relators

In Chapter 2, we explained why we needed to work in a category **Coc** (**C**) whose objects are ω-cocontinuous endofunctors on a simpler category **C**. The purpose of this chapter is to explain what happens when we change our choice of **C** from **Fun**, the category of sets and total functions, to **Rel**, the category of sets and relations. Relations are useful in program calculation because the specification of a program may be non-deterministic or partial even though the implementation cannot be. However, the move to relations requires that we rewrite most of the foundation material from the previous chapters, and because of this, the next three chapters directly depend on this chapter.

Section 5.1 describes the category **Rel** and augments it with the operators of the relational calculus, to form a special sort of category called an allegory. Section 5.2 defines endorelators, which are endofunctors with a certain property, and describes the category **Cor** (**Rel**) whose objects are ω-cocontinuous endorelators on **Rel**. It is this category that we shall use for the rest of this thesis. We choose it in preference to the category **Coc** (**Rel**), whose objects are endofunctors that are not necessarily endorelators, because endorelators are canonical extensions.

Section 5.3 writes maps as efficient folds and shows that nested functors on **Fun** extend to nested relators on **Rel**, by showing that initial algebras in **Coc** (**Fun**) are also initial algebras in **Cor** (**Rel**). This is the key result of the chapter. Section 5.4 gives relational versions of the fusion laws for generalised folds. Section 5.5 describes the hylomorphism theorem for standard folds and considers whether it can be extended to folds on nested datatypes.

Finally, Sections 5.6 and 5.7 contain preliminary material for Chapter 8. Section 5.6 shows that efficient fold operators take injective relations to injective relations and Section 5.7 derives a fold-fusion law for efficient reductions.

5.1 The category of relations

The category **Rel** has sets as objects and relations as arrows. Relations shall be represented as subsets of the cartesian product. For any two relations R and S, we define the meet $R \cap S$ to be the intersection of the subsets of the cartesian product that correspond to R and S. The inclusion ordering on relations $\subseteq$ is defined similarly. We shall write $x\,R\,y$ as shorthand for $(x, y) \in R$ and interpret this to mean that an input y is mapped by a relation R to an output x. The converse of a relation R is denoted R° and defined by $x\,R^\circ\,y \;\equiv\; y\,R\,x$.

The three operators $\cap$, $\subseteq$ and $(-)^\circ$ make **Rel** a special kind of category known as an allegory. Formally, an *allegory* is a category with three extra operators:

- a partial order $\subseteq$ on arrows of the same type, and
- a binary meet operator $\cap$ that takes two arrows R and S of the same type to a third arrow $R \cap S$ of that type, and
- a unary operator $(-)^\circ$ that takes an arrow R of type $A \to B$ to an arrow R°, called the *converse* of R, with type $B \to A$.

These three operators must satisfy six axioms. The binary operator $\cap$ is defined by the universal property

$$X \subseteq R \cap S \;\equiv\; X \subseteq R \quad \text{and} \quad X \subseteq S$$

The partial order $\subseteq$ is monotonic with respect to composition:

$$S_1 \subseteq S_2 \quad \text{and} \quad T_1 \subseteq T_2 \;\Rightarrow\; S_1 \cdot T_1 \subseteq S_2 \cdot T_2$$

The following three axioms state respectively that converse is order-preserving, contravariant with composition and an involution.

$$\begin{aligned} R \subseteq S &\;\Rightarrow\; R^\circ \subseteq S^\circ \\ (R \cdot S)^\circ &\;=\; S^\circ \cdot R^\circ \\ (R^\circ)^\circ &\;=\; R \end{aligned}$$

The antisymmetry of $\subseteq$ can be used to strengthen the first axiom of converse to

$$R = S \;\Rightarrow\; R^\circ = S^\circ$$

The fifth axiom can now be used to strengthen this further to

$$R = S \;\equiv\; R^\circ = S^\circ$$

Therefore we preserve the truth of any equality or inequality if we take the converse of both sides. The three operators are connected by the following modular law.

$$(R \cdot S) \cap T \;\subseteq\; R \cdot (S \cap (R^\circ \cdot T))$$

Although we shall not need to use this law, it must be included as a sixth axiom of allegories as it cannot be proven from the first five. To illustrate these axioms, we show that converse distributes over meet:

$$(R \cap S)^\circ \;=\; R^\circ \cap S^\circ$$

The proof is as follows:

$$\begin{array}{ll}
 & X \subseteq (R \cap S)^\circ \\
\equiv & \quad \{ \text{ taking converse of both sides } \} \\
 & X^\circ \subseteq R \cap S \\
\equiv & \quad \{ \text{ universal property of meet } \} \\
 & X^\circ \subseteq R \quad \text{and} \quad X^\circ \subseteq S \\
\equiv & \quad \{ \text{ taking converse of both sides } \} \\
 & X \subseteq R^\circ \quad \text{and} \quad X \subseteq S^\circ \\
\equiv & \quad \{ \text{ universal property of meet } \} \\
 & X \subseteq R^\circ \cap S^\circ
\end{array}$$

A union operator $\cup$ can be defined on relations in a manner similar to the meet operator. Its universal property is also similar:

$$R \cup S \subseteq X \;\equiv\; R \subseteq X \quad \text{and} \quad S \subseteq X$$

Composition distributes over union.

$$(R_1 \cup R_2) \cdot S \;=\; (R_1 \cdot S) \cup (R_2 \cdot S)$$

The allegory **Rel** is *locally complete*, which means that the n-ary generalisation of $\cup$ is defined for all sets of arrows of the same type.

A function f can be shunted from one side of an inequation to the other, in accordance with the shunting rules below.

$$\begin{array}{rcl} f \cdot R \subseteq S & \equiv & R \subseteq f^\circ \cdot S \\ R \cdot f^\circ \subseteq S & \equiv & R \subseteq S \cdot f \end{array}$$

5.1.1 Relational product

The allegory **Rel** has a particular property that will cause some difficulties for us: products in the allegory are isomorphic to coproducts and therefore indistinguishable from them. This is because reversing every arrow in the universal property of products gives the universal property of coproducts and a reverse arrow exists for every arrow in **Rel**. Therefore we must define a new relational product. Its fork operator must satisfy

$$(a, b)\, \langle R, S\rangle\, c \quad \equiv \quad a\, R\, c \quad \text{and} \quad b\, S\, c$$

When R is a function this operator behaves exactly like the fork operator of **Fun**. The right-hand side is equivalent to

$$((a, b)\, \mathit{outl}^\circ\, a \quad \text{and} \quad a\, R\, c) \quad \text{and} \quad ((a, b)\, \mathit{outr}^\circ\, b \quad \text{and} \quad b\, S\, c)$$

Therefore we define the relational fork operator by

$$\langle R, S\rangle \quad = \quad (\mathit{outl}^\circ \cdot R) \quad \cap \quad (\mathit{outr}^\circ \cdot S)$$

Since we shall not use categorical product again, no ambiguity is introduced when we recycle its notation to denote relational product. Note however that $\mathit{outl} \cdot \langle R, \{\}\rangle \neq R$ so relational product and categorical product are different. Furthermore, relational product does not have a universal property. The action of $\times$ on relations is given by

$$R \times S \quad = \quad \langle R \cdot \mathit{outl}, S \cdot \mathit{outr}\rangle$$

It is shown in [BdM97] that $\times$ is a bifunctor. Finally, we note the following absorption law.

$$(R \times S) \cdot \langle X, Y\rangle \quad = \quad \langle R \cdot X, S \cdot Y\rangle$$

5.2 A category of lax natural transformations

The category that we shall use for the rest of this thesis is **Cor** (**Rel**), the category whose objects are ω-cocontinuous endorelators on **Rel** and whose arrows are lax natural transformations. The term ω-cocontinuous is defined in [MG01] but it is sufficient to note here that all nested functors are ω-cocontinuous. We shall call **Coc** (**Fun**) the endofunctor category and **Cor** (**Rel**) the endorelator category. We use **Cor** (**Rel**) instead of **Coc** (**Rel**) because relators are canonical extensions of functors so **Cor** (**Rel**) has sufficient objects, and also because we need to use properties of endorelators in our proofs, so **Coc** (**Rel**) has too many objects. We shall now explain exactly what are endorelators and lax natural transformations.

5.2.1 Relators

Recall the definition of the relational product bifunctor on arrows of **Rel**.

$$R \times S = \langle R \cdot outl, S \cdot outr \rangle$$

This bifunctor coincides with the categorical product bifunctor on **Fun** when R and S are total functions, simply because the corresponding fork operations for **Fun** and **Rel** also coincide for total functions. We say that the relational product bifunctor is a *relational extension* of the categorical product bifunctor. However, it is possible that other bifunctors are also relational extensions of the categorical product bifunctor. To make the relational product bifunctor be a canonical extension, we require that it be a relator. A *relator* is a functor F on relations that commutes with converse, that is,

$$F\ (R^\circ) \quad = \quad (F\ R)^\circ$$

It is shown in [BdM97] that relators preserve total functions (so they are relational extensions) and that if two relators are relational extensions of the same functor then they are equal (so they are canonical extensions). For example, the relational product bifunctor is a relator because it commutes with converse [BdM97], that is

$$R^\circ \times S^\circ \quad = \quad (R \times S)^\circ$$

Intuitively this means that undoing R on the left component of a pair and undoing S on the right undoes the effect of doing R on the left and doing S

on the right. From this alone we can tell that the relational product bifunctor is a canonical relational extension of the categorical product bifunctor. Similarly, the categorical coproduct bifunctor in **Fun** extends to the categorical coproduct bifunctor in **Rel**. In fact, all polynomial functors extend to relators. In the next section we shall show that nested functors extend to relators, but first we shall introduce lax natural transformations.

5.2.2 Lax natural transformations

In Chapter 2, we discovered that *hflatnest* satisfies the following naturality property, for all functions $f : A \to B$,

$$\mathit{List}\, f \cdot \mathit{hflatnest}_A \quad = \quad \mathit{hflatnest}_B \cdot \mathit{Nest}\, f$$

However, we now want *Nest* and *List* to be endorelators on **Rel** so a more useful property would be that for all relations $R : A \to B$, we have

$$\mathit{List}\, R \cdot \mathit{hflatnest}_A \quad = \quad \mathit{hflatnest}_B \cdot \mathit{Nest}\, R$$

This property says that mapping a relation over a nest before flattening it has the same effect as flattening the nest first and then mapping the relation over the list produced. It is tempting to think that we can always generalise from a function f to any relation R the naturality condition of a natural transformation ϕ. However, while the generalisation is correct for *hflatnest* it turns out that we must impose severe restrictions on both ϕ and R to maintain equality in general.

First, if ϕ copies elements then R must be deterministic for the equality to hold. Suppose that $\phi = \mathit{duplic}$, where *duplic* was defined in Chapter 2, and that R is the relation that non-deterministically adds one or subtracts one from an integer. Then the relation $\mathit{Pair}\, R \cdot \mathit{duplic}_{\mathit{Int}}$ maps each input to twice as many outputs as the relation $\mathit{duplic}_{\mathit{Int}} \cdot R$, so instead of equality we have (using $\supseteq$ to denote the reverse of $\subseteq$)

$$\mathit{Pair}\, R \cdot \mathit{duplic}_{\mathit{Int}} \supseteq \mathit{duplic}_{\mathit{Int}} \cdot R$$

Secondly, if ϕ removes elements then R must not be partial. Suppose that $\phi = \mathit{outl}$ and $R = \{(0,0)\}$, that is, the identity function restricted to the value 0. Then the relation $R \cdot \mathit{outl}_{\mathit{Int},\mathit{Int}}$ is defined on the input pair $(0,1)$

but the relation $outl_{Int,Int} \cdot (R \times R)$ is not, and as before we must write an inequality.

$$R \cdot outl_{Int,Int} \;\supseteq\; outl_{Int,Int} \cdot (R \;\times\; R)$$

However, if ϕ neither copies nor removes elements then the equality holds for all relations R. This case, illustrated by *hfoldnest*, is described as a proper natural transformation. More often, there is an inequality for all relations R and this more general case is described as a lax natural transformation.

A *lax natural transformation ϕ from F to G*, denoted by $\phi : F \hookrightarrow G$, is a collection of arrows $\phi_A : F\,A \rightarrow G\,A$ such that for all $R : A \rightarrow B$,

$$G\,R \cdot \phi_A \;\;\supseteq\;\; \phi_B \cdot F\,R$$

A *proper natural transformation ϕ from F to G*, denoted by $\phi : F \stackrel{.}{\rightarrow} G$, is defined similarly but with an equality in place of the inequality in the naturality condition. It is shown in [BdM97] that when F and G are relators, the naturality condition of lax natural transformations is equivalent to the condition that for functions $f : A \rightarrow B$,

$$G\,f \cdot \phi_A \;\;=\;\; \phi_B \cdot F\,f$$

5.2.3 Further structure of the endorelator category

The operators $(-)^{\circ}$, $\cup$, $\cap$ and $\subseteq$ on arrows of **Rel** all lift to arrows of **Coc** (**Rel**) and hence **Cor** (**Rel**) in the obvious way. We know that categorical product in **Fun** extends to relational product in **Rel** and that categorical coproduct in **Fun** extends to categorical coproduct in **Rel**. These two statements remain true after lifting to the categories **Nat** (**Fun**) and **Nat** (**Rel**) [MG01], and even after restricting attention to the smaller categories **Coc** (**Fun**) and **Cor** (**Rel**). The fusion laws of coproduct and the absorption law both lift to lax natural transformations.

5.3 Nested functors extend to relators

We now plan to represent type constructors using relators on **Rel** instead of functors on **Fun**. This change is necessary because our map operators must be defined on relations rather than on total functions alone. We shall use

the same identifier for both the original functor and the relator that replaces it. We can do this without fear of ambiguity because there is at most one relator extending each functor [BdM97]. However, first we must prove that there is *at least one* nested relator extending each nested functor.

First we show that initial algebras in the endofunctor category are also initial in the endorelator category. From this, we can prove that efficient fold operators constructed in **Coc** (**Fun**) extend to efficient fold operators constructed in **Cor** (**Rel**) that preserve total functions. By writing maps as efficient folds, we immediately conclude that nested functors on **Fun** extend to nested functors on **Rel**.

However, for all this to work, we must show that map operations for nested endofunctors constructed in **Coc** (**Fun**) and **Coc** (**Rel**) are endorelators so that they can also be constructed in **Cor** (**Rel**). We need to check that **Coc** (**Rel**) has initial algebras because we use efficient folds in our proof. Fortunately, it does because **Rel** has empty sets as initial objects, all finite coproducts and it is also ω-cocontinuous, as shown in [MG01]. It is clear that **Rel** has all relational products because they are defined by a simple equation rather than a universal property. Note however that relational product is defined in terms of categorical product so it is important that the latter exists for **Rel**.

5.3.1 Power allegories

In order to prove that initial algebras in the endofunctor category are also initial algebras in the endorelator category, we need to use the property of **Rel** of it being a power allegory. Informally, this means that there is an isomorphism between relations and set-valued total functions.

More formally, we have

- for each object A, an object $\mathsf{P}\,A$, and
- for each arrow $R : A \to B$, a function $\Lambda R : A \to \mathsf{P}\,B$, and
- an arrow $\in : \mathsf{P}\,A \to A$, and

- for all $R : A \to B$ and $f : A \to \mathsf{P}\, B$, the universal property

$$f \;=\; \Lambda R \;\;\equiv\;\; \in \cdot f \;=\; R$$

Here P is the power set endorelator on **Rel** defined pointwise on sets by

$$\mathsf{P}\, A \;\;=\;\; \{x \mid x \subseteq A\}$$

and on relations by

$$x\;(P\,R)\,y \;\;\equiv\;\; (\forall a \in x : \exists b \in y : a\,R\,b) \text{ and } (\forall b \in y : \exists a \in x : a\,R\,b)$$

The right-hand side says that every element in x is related to some element in y and vice versa. It follows from the categorical definition of P (see [BdM97]) that $\in$ is a lax natural transformation of type $\mathsf{P} \stackrel{.}{\hookrightarrow} Id$.

Taking $f = \Lambda R$ in the universal property gives a *cancellation law*.

$$\in \cdot \Lambda R \;\;=\;\; R$$

Taking $f = \Lambda\, S$ gives $\in \cdot \Lambda S = R$ and then $S = R$ by the cancellation law so Λ is the isomorphism between relations and set-valued functions.

$$R \;=\; S \;\;\equiv\;\; \Lambda R \;=\; \Lambda S$$

Using the cancellation law and the universal property together gives a *fusion law*. For a function f, we have

$$\Lambda(R \cdot f) \;\;=\;\; \Lambda R \cdot f$$

5.3.2 Endofunctor categories based on power allegories

In fact, we wish to use Λ and $\in$ lifted to lax natural transformations. Whilst it is not obvious that we can do this, we can certainly lift the laws above to collections of arrows in **Rel**.

$$\begin{aligned}
\theta \;=\; \Lambda\eta \;\;&\equiv\;\; \in \cdot\, \theta \;=\; \eta \\
\in \cdot \Lambda\eta \;\;&=\;\; \eta \\
\eta \;=\; \eta' \;\;&\equiv\;\; \Lambda\eta \;=\; \Lambda\eta' \\
\Lambda(\eta \cdot \theta) \;\;&=\;\; \Lambda\eta \cdot \theta
\end{aligned}$$

Here, θ denotes a collection of functions and both η and η' denote collections of relations. We already know that $\in$ is a lax natural transformation so we only need to show that Λ takes lax natural transformations to lax natural transformations. Suppose η is a lax natural transformation of type $F \stackrel{\cdot}{\hookrightarrow} G$, where F and G are relators. Then we must show for all functions f that

$$\mathsf{P}\ (G\,f) \cdot (\Lambda\,\eta)_A \quad = \quad (\Lambda\,\eta)_B \cdot F\,f$$

We reason

$$\begin{array}{ll} & \mathsf{P}\ (G\,f) \cdot (\Lambda\eta)_A \\ = & \qquad \{\ \text{see below}\ \} \\ & \Lambda(\eta_B \cdot F\,f) \\ = & \qquad \{\ \text{fusion law of } \Lambda\text{; relators preserve functions}\ \} \\ & \Lambda(\eta_B) \cdot F\,f \\ = & \qquad \{\ \text{definition of lifted } \Lambda\ \} \\ & (\Lambda\,\eta)_B \cdot F\,f \end{array}$$

By the universal property of Λ, the first step is equivalent to

$$\in \cdot\, \mathsf{P}\ (G\,f) \cdot (\Lambda\,\eta)_A \quad = \quad \eta_B \cdot F\,f$$

We prove this as follows.

$$\begin{array}{ll} & \eta_B \cdot F\,f \\ = & \qquad \{\ \text{naturality of } \eta\ \} \\ & G\,f \cdot \eta_A \\ = & \qquad \{\ \text{cancellation law of } \Lambda\ \} \\ & G\,f \cdot \in \cdot \Lambda(\eta_A) \\ = & \qquad \{\ \text{naturality of } \in\ \} \\ & \in \cdot\, \mathsf{P}\ (G\,f) \cdot \Lambda(\eta_A) \\ = & \qquad \{\ \text{definition of lifted } \Lambda\ \} \\ & \in \cdot\, \mathsf{P}\ (G\,f) \cdot (\Lambda\eta)_A \end{array}$$

5.3.3 Initial algebras extend to endorelator category

Now we finally prove that initial algebras in the endofunctor category are also initial algebras in the endorelator category. Let F be a polynomial hofunctor. Then F is also an endofunctor on the endorelator category so it preserves lax natural transformations. Polynomial functors extend to relators. Higher-order polynomial functors extend to higher-order polynomial relators, or *horelators*, which preserve functional lax natural transformations. An example will help us convince ourselves of this. If we apply the hofunctor *NestF* to lax natural transformations, we get (using the notation introduced in Section 2.5.7)

$$\mathit{NestF}\,\eta \quad = \quad K_{K_1}\,\eta + K_{\mathit{Id}}\,\eta \times (\mathit{Id} \star K_{\mathit{Pair}})\eta$$

Simplifying this a little gives

$$\mathit{NestF}\,\eta \quad = \quad \mathit{id}_{K_1} + \mathit{id} \times \eta_{\mathit{Pair}}$$

We know that *NestF* preserves lax natural transformations but we can also see now that it must preserve functional lax natural transformations. The binary relators $+$ and $\times$ preserve functions so when they are lifted to collections of arrows they preserve functional collections of arrows. Finally, by examining the definitions of composition and horizontal composition, we can satisfy ourselves that all polynomial hofunctors extend to horelators.

Now for the proof.

$$\begin{array}{ll}
 & X \cdot \alpha = \eta \cdot F\,X \\
\equiv & \quad \{\ \Lambda \text{ is an isomorphism }\} \\
 & \Lambda(X \cdot \alpha) = \Lambda(\eta \cdot F\,X) \\
\equiv & \quad \{\ \text{cancellation law of } \Lambda\ \} \\
 & \Lambda(X \cdot \alpha) = \Lambda(\eta \cdot F\,(\in \cdot \Lambda X)) \\
\equiv & \quad \{\ F \text{ is a horelator; fusion law of } \Lambda \text{ on both sides }\} \\
 & \Lambda X \cdot \alpha = \Lambda(\eta \cdot F\ \in) \cdot F\,(\Lambda X) \\
\equiv & \quad \{\ \text{universal property of simple folds }\} \\
 & \Lambda X = (\!|\Lambda(\eta \cdot F\ \in)|\!)_F \\
\equiv & \quad \{\ \text{cancellation law of } \Lambda\ \} \\
 & X = \in \cdot (\!|\Lambda(\eta \cdot F\ \in)|\!)_F
\end{array}$$

5.3.4 Efficient fold operators preserve total functions

Since initial algebras in the endofunctor category are also initial algebras in the endorelator category, an efficient fold constructed in the endofunctor category is defined by the same equation as an efficient fold constructed in the endorelator category from the same functional arguments. However, this equation is a universal property with a unique solution, so the two efficient folds are equal and the efficient fold operator of **Cor** (**Rel**) extends the efficient fold operator of **Coc** (**Fun**). In other words, the efficient fold operator of **Cor** (**Rel**) preserves deterministic total lax natural transformations, and polymorphic total functions in particular.

5.3.5 Map operations are efficient folds

In Chapter 2, we discovered that map operations are not simple folds because they have the wrong type. In Chapter 3, we discovered that they cannot be expressed as generalised folds either, because generalised folds are themselves defined in terms of map operations. However, an efficient fold is equal to a generalised fold after a map operation and the identity function is a generalised fold so we have

$$\begin{array}{rcl} T & : & (a \to b) \to (T\ a \to T\ b) \\ T\,R & = & \{\![\alpha \,\|\, ident \,\|\, R]\!\}_F \end{array}$$

Recall that $ident_i = id_{F_i\ T}$. For regular relators, we would write the map operation as a standard fold in the same way as we did with lists in Chapter 2. However, this equation would be a definition of the map operation rather than a property it satisfies as we have here and it would then be necessary to prove that the map operation preserves identities and composition. We do not need to do this because the objects of the endofunctor and endorelator categories are functors by definition.

5.3.6 The functor Nest commutes with converse

Our final task is to show for all nested T that

$$T\ (R^\circ) \;=\; (T\,R)^\circ$$

We need to do this so that we can be sure that map operations constructed in **Coc** (**Fun**) and **Coc** (**Rel**) can also be constructed in **Cor** (**Rel**). Note

that the converse of lax natural transformations is defined by lifting so this equation extends at once to lax natural transformations. For regular T, we prove the equation by writing the map operation as a standard fold and then applying the universal property of standard folds [BdM97]. To mimic this approach, we shall write the map operation as an efficient fold and apply the universal property of efficient folds. We start by rehearsing our proof for the case of $T = Nest$.

$$
\begin{array}{ll}
 & (Nest\, R)^\circ = Nest(R^\circ) \\
\equiv & \quad \{ \text{ maps are efficient folds; } \mathbf{let}\ \chi\,(R^\circ) = (Nest\, R)^\circ \ \} \\
 & \chi\,(R^\circ) = \{\![\alpha \mid id \mid R^\circ]\!\}_F \\
\equiv & \quad \{ \text{ universal property of efficient folds } \} \\
 & \chi\ (R^\circ) \cdot \alpha = \alpha \cdot Base\ (R^\circ, \chi\ (id \cdot Pair\ (R^\circ))) \\
\equiv & \quad \{ \ Pair \text{ is a relator } \} \\
 & \chi\ (R^\circ) \cdot \alpha = \alpha \cdot Base\ (R^\circ, \chi\ ((Pair\, R)^\circ)) \\
\equiv & \quad \{ \text{ definition of } \chi \ \} \\
 & (Nest\, R)^\circ \cdot \alpha = \alpha \cdot Base\ (R^\circ, (Nest\ (Pair\, R))^\circ) \\
\equiv & \quad \{ \text{ converse is order-preserving and contravariant } \} \\
 & \alpha^\circ \cdot Nest\, R = (Base\ (R^\circ, (Nest\ (Pair\, R))^\circ))^\circ \cdot \alpha^\circ \\
\equiv & \quad \{ \text{ shunting } \alpha\text{; } Base \text{ is a relator } \} \\
 & Nest\, R \cdot \alpha = \alpha \cdot Base\ (R,\ Nest\ (Pair\, R)) \\
\equiv & \quad \{ \text{ naturality of } \alpha \ \} \\
 & \text{true}
\end{array}
$$

5.3.7 All nested functors commute with converse

Now we generalise the proof above to arbitrary nested T.

$$
\begin{array}{rcl}
T & \approx & F\, T \\
F\, X & = & B \cdot \langle Id, X \cdot F_1\, X \rangle
\end{array}
$$

As usual, there is an unspecified number of subsidiary hofunctors F_i each of the form, for some polynomial bifunctor B and hofunctor F_j,

$$
F_i\, X \quad = \quad B_i \cdot \langle Id, X \cdot F_j\, X \rangle
$$

As before with nests, we shall define the mapping on relations χ by $\chi\ (R^\circ) = (T\ R)^\circ$. When we apply the universal property this time, we get the following two proof obligations.

$$\begin{array}{rcl} \chi\ (R^\circ) & = & \{\![\alpha \parallel \mathit{ident} \parallel R^\circ]\!\}_F \\ \xi_i\ (R^\circ) & = & B_i\ (R^\circ, \chi\ (\xi_j\ (R^\circ))) \end{array}$$

We derive a definition of ξ as a missing step when proving the first of these.

$$\begin{array}{ll} & \chi\ (R^\circ) = \{\![\alpha \parallel \mathit{ident} \parallel R^\circ]\!\}_F \\ \equiv & \quad \{ \text{ universal property of efficient folds } \} \\ & \chi\ (R^\circ) \cdot \alpha = \alpha \cdot B\ (R^\circ, \chi\ (\xi_1\ (R^\circ))) \\ \equiv & \quad \{ \text{ deriving } \xi \text{ (see below) } \} \\ & \chi\ (R^\circ) \cdot \alpha = \alpha \cdot B\ (R^\circ, \chi\ ((F_1\ T\ R)^\circ)) \\ \equiv & \quad \{ \text{ definition of } \chi \ \} \\ & (T\ R)^\circ \cdot \alpha = \alpha \cdot B\ (R^\circ, (T\ (F_1\ T\ R))^\circ) \\ \equiv & \quad \{ \text{ applying converse to both sides } \} \\ & \alpha^\circ \cdot T\ R = (B\ (R^\circ, (T\ (F_1\ T\ R))^\circ)^\circ) \cdot \alpha^\circ \\ \equiv & \quad \{ \text{ shunting } \alpha^\circ \text{ twice; } B \text{ is a relator } \} \\ & T\ R \cdot \alpha = \alpha \cdot B\ (R,\ T\ (F_1\ T\ R)) \\ \equiv & \quad \{ \text{ naturality of } \alpha \ \} \\ & \text{true} \end{array}$$

The definition of ξ derived above is, for all i

$$\xi_i\ (R^\circ) \quad = \quad (F_i\ T\ R)^\circ$$

Now we use this to check the second condition; let F_i be defined as above.

$$\begin{array}{ll} & \xi_i\ (R^\circ) \\ = & \quad \{ \text{ definition } \xi \ \} \\ & (F_i\ T\ R)^\circ \\ = & \quad \{ \text{ definition of } F_i \ \} \\ & (B_i\ (R, T\ (F_j\ T\ R)))^\circ \\ = & \quad \{ \text{ relator } B_i \ \} \\ & B_i\ (R^\circ, (T\ (F_j\ T\ R))^\circ) \end{array}$$

$$= \qquad \{ \text{ definition of } \chi \ \}$$
$$B_i\ (R^\circ, \chi\ ((F_j\ T\ R)^\circ))$$

5.3.8 Nested relators are monotonic

The allegory **Rel** has the special property that all functors on it that commute with converse are also monotonic (to be defined) so all nested functors are monotonic. This property follows from **Rel** being tabular, which shall be defined in the next section. A functor F is *monotonic* if

$$R \subseteq S \quad \Rightarrow \quad F\,R \subseteq F\,S$$

Suppose we say that a relation is *better than* another relation if it is defined on at least the same inputs with at least the same outputs for each input. Then the monotonicity condition says that if doing S is better than doing R, then doing S on each element in an structure is better than doing R on each element.

5.3.9 Generalising to other allegories

We have implicitly used two properties of **Rel** while reasoning in this section: the property that **Rel** is a power allegory and the property that **Rel** is a tabular allegory. The first of these has already been defined. The tabularity property is that for every $R : A \rightarrow B$ there is a set C and two functions $f : C \rightarrow A$ and $g : C \rightarrow B$ such that

$$R \;=\; g \cdot f^\circ \quad \text{and} \quad (f^\circ \cdot f) \,\cap\, (g^\circ \cdot g) \;=\; id_C$$

In fact, C is a subset of $A \times B$ and f and g are the projections $outl_{A,B}$ and $outr_{A,B}$.

However, there is a case for using as few properties of **Rel** as possible so that we can generalise this section to as many endofunctor categories as possible. It should be pointed out however that this is primarily an aesthetic consideration as we are not interested in any other endofunctor categories. For example, Hoogendijk uses neither property to reason with standard folds and regular datatypes so his results, analogous to ours, extend to any allegory [Hoo97].

It is the tabularity assumption that allows us to conclude that nested functors are monotonic from the fact that they commute with converse. We have to prove this directly if we drop the assumption. Adapting the proof for regular datatypes in [Hoo97], we could try using the forthcoming fold-fusion law for efficient folds to show given $R \subseteq S$ that

$$id \cdot \{\![\alpha \parallel ident \parallel R]\!\}_F \subseteq \{\![\alpha \parallel ident \parallel S]\!\}_F$$

However, this fold-fusion law is proved later using monotonicity so we cannot use it to prove monotonicity itself.

If we drop the assumption that **Rel** is a power allegory then we must find another way to show that efficient fold operators constructed in **Cor** (**Rel**) preserve total functions. Now, a relation is a total function if and only if it is both simple and total. A relation R is *simple* if $R \cdot R^\circ \subseteq id$ and *total* if $id \subseteq R^\circ \cdot R$. To prove that the efficient fold operator preserves simple relations, we just substitute R° for R in the forthcoming injectivity proof. That this is sufficient will be clear when we define injectivity.

To prove that the operator preserves functions, it remains to show that it preserves total relations. Hoogendijk [Hoo97] shows this using fold-fusion for standard folds but when we try to repeat his proof for generalised folds, we get (for nests)

$$(\![\alpha \mid id]\!)_{NestF} \subseteq (\![f \mid g]\!)_{NestF}{}^\circ \cdot (\![f \mid g]\!)_{NestF}$$

The typing rules of the fold-fusion law make it clear that the fold-fusion law yields a simple fold only if it is given a simple fold so our proof would be limited to simple folds. The case of efficient folds is similar.

In conclusion, we only know how to extend the material in this section to **Cor** (**C**) where **C** is a tabular power allegory, whereas analogous results extend to any allegory **C**.

5.4 Relational fusion laws

5.4.1 Knaster-Tarski theorem

The initial algebra of a hofunctor F is an isomorphism, by Lambek's theorem, and therefore a function so we can shunt it to rearrange the universal property

of the generic generalised fold operator. For some Ψ we get

$$h = (\![f \parallel \gamma]\!)_F \quad \equiv \quad h = \Psi\ (f, \gamma, h) \cdot \alpha^\circ$$

Now $(\![f \parallel \gamma]\!)_F$ is the unique solution for, and therefore both the least and greatest solution for, the equation on the right, which has the form $X = \theta\, X$ where

$$\theta\, X \quad = \quad \Psi\ (f, \gamma, X) \cdot \alpha^\circ$$

It is easy to verify that θ is a monotonic mapping because polynomial and nested relators are monotonic and composition is monotonic. The Knaster-Tarski theorem [BdM97] makes two claims. The first claim is that the greatest solution of $X = \theta\, X$ exists, for monotonic θ, and is also the greatest solution of $X \subseteq \theta\, X$. The second claim is that the least solution of $\theta\, X = X$ exists, for monotonic θ, and is also the least solution of $\theta\, X \subseteq X$. The theorem is restricted to complete lattices, but the endorelator category clearly is a complete lattice with $\subseteq$ as the ordering, $\cap$ as the meet operator and $\cup$ as the join operator.

Now we can rephrase the first claim of Knaster-Tarski to say that the unique solution of $X = \theta\, X$, which is $(\![f \parallel \gamma]\!)_F$, includes any solution of $X \subseteq \theta\, X$:

$$h \subseteq (\![f \parallel \gamma]\!)_F \quad \Leftarrow \quad h \subseteq \Psi\ (f, \gamma, h) \cdot \alpha^\circ$$

We can also rephrase the second claim to say that the unique solution of $X = \theta\, X$ is included in any solution of $\theta\, X \subseteq X$.

$$(\![f \parallel \gamma]\!)_F \subseteq h \quad \Leftarrow \quad \Psi\ (f, \gamma, h) \cdot \alpha^\circ \subseteq h$$

5.4.2 Relational fold-fusion law

The generic fold-fusion law of Chapter 4 also holds in the endorelator category but it is more useful when stated using the operator $\subseteq$. To prove this alternative form, we must retrace the original derivation by Bird and Paterson [BP99b] and then retrace our own derivation from Chapter 4, replacing equations with inequations in both.

Here is one half of the universal property of generalised folds.

$$h = (\![f \parallel \gamma]\!)_F \quad \Leftarrow \quad h \cdot \alpha = \Psi\ (f, \gamma, h)$$

Recall that Bird and Paterson instantiate this to

$$k \cdot (\!|f \parallel \gamma|\!)_F = (\!|f' \parallel \gamma'|\!)_F \quad \Leftarrow \quad k \cdot (\!|f \parallel \gamma|\!)_F \cdot \alpha = \Psi\ (f', \gamma', k \cdot (\!|f \parallel \gamma|\!)_F)$$

Then they prove the right-hand equation, deriving the following two conditions as steps:

$$\begin{aligned} k \cdot f \quad &= \quad f' \cdot B\ (id, k \cdot R\, p) \\ M\ p \cdot \Phi_1\ (\gamma,\ (\!|f \parallel \gamma|\!)_F) \quad &= \quad \Phi_1\ (\gamma',\ k \cdot (\!|f \parallel \gamma|\!)_F) \end{aligned}$$

Instead, we can shunt α° in the first claim of Knaster-Tarski and instantiate it to

$$k \cdot (\!|f \parallel \gamma|\!)_F \subseteq (\!|f' \parallel \gamma'|\!)_F \quad \Leftarrow \quad k \cdot (\!|f \parallel \gamma|\!)_F \cdot \alpha \subseteq \Psi\ (f', \gamma', k \cdot (\!|f \parallel \gamma|\!)_F)$$

Then we prove the right-hand inequation, deriving the following two conditions:

$$\begin{aligned} k \cdot f \quad &\subseteq \quad f' \cdot B\ (id, k \cdot R\, p) \\ M\ p \cdot \Phi_1\ (\gamma,\ (\!|f \parallel \gamma|\!)_F) \quad &\subseteq \quad \Phi_1\ (\gamma',\ k \cdot (\!|f \parallel \gamma|\!)_F) \end{aligned}$$

Now we can repeat our derivation of the improved fold-fusion law with no fold or map operations in its conditions. The second condition above follows from the collection of conditions (for all i and appropriate j)

$$M\ p_i \cdot \gamma_i \quad \subseteq \quad \gamma'_i \cdot B_i\ (id, k \cdot R\, p_j)$$

The law we have derived is as follows

Relational generic fold-fusion law If we have an indexed collection of functions p such that for all i with $F_i\, X = B_i \cdot \langle Id, X \cdot F_j\, X \rangle$,

$$M\ p_i \cdot \gamma_i \quad \subseteq \quad \gamma'_i \cdot B_i\ (id, k \cdot R\, p_j)$$

and

$$k \cdot f \quad \subseteq \quad f' \cdot B\ (id, k \cdot R\ p_1)$$

then we can conclude that

$$k \cdot (\!|f \parallel \gamma|\!)_F \quad \subseteq \quad (\!|f' \parallel \gamma'|\!)_F$$

5.4.3 Relational map-fusion law

To derive the map-fusion law, Bird and Paterson instantiate one half of the universal property to

$$
\begin{aligned}
&(\!|f \parallel \gamma|\!)_F \cdot T\,k = (\!|f' \parallel \gamma'|\!)_F \\
&\qquad \Leftarrow \quad (\!|f \parallel \gamma|\!)_F \cdot T\,k \cdot \alpha = \Psi\ (f', \gamma', (\!|f \parallel \gamma|\!)_F \cdot T\,k)
\end{aligned}
$$

They derive the following conditions when proving the right-hand equality.

$$
\begin{aligned}
f' &= f \cdot B\ (k, id) \\
\Phi_1\ (\gamma,\ (\!|f \parallel \gamma|\!)_F) \cdot F_1\ T\,k &= k_{F_1\,R} \cdot \Phi_1\ (\gamma',\ (\!|f \parallel \gamma|\!)_F \cdot T\,k)
\end{aligned}
$$

To derive a relational version of the map-fusion law, we shunt α° in the second claim and instantiate it to

$$
\begin{aligned}
&(\!|f' \parallel \gamma'|\!)_F \subseteq (\!|f \parallel \gamma|\!)_F \cdot T\,k \\
&\qquad \Leftarrow \quad \Psi\ (f', \gamma', (\!|f \parallel \gamma|\!)_F \cdot T\,k) \subseteq (\!|f \parallel \gamma|\!)_F \cdot T\,k \cdot \alpha
\end{aligned}
$$

Then we redo Bird and Paterson's derivation to get the two conditions above stated as inequalities and redo our derivation with the second condition to give the following law.

Relational generic map-fusion law If we have

$$k \cdot \gamma_i' \subseteq \gamma_i \cdot B_i\ (k, id)$$

and

$$f \cdot B\ (k, id) \ \subseteq\ f'$$

then we can conclude

$$(\!|f \cdot B\ (k, id) \parallel \gamma'|\!)_F \ \subseteq\ (\!|f \parallel \gamma|\!)_F \cdot T\,k$$

Both this law and the fold-fusion law have variants where the $\subseteq$'s are replaced by $\supseteq$'s. However, we shall not need either variant.

5.5 Hylomorphism theorem

An *unfold* is the converse of a fold and it is typically used for building a data structure, whereas a fold typically consumes a data structure. A *hylomorphism* is a fold preceded by an unfold. A *hylomorphism theorem* writes a hylomorphism as a least fixed point. Such a theorem can be used to eliminate intermediate data structures in programs. A hylomorphism theorem for standard folds is proved in [BdM97]. By lifting the proof to an endofunctor category we can show that there is a hylomorphism theorem for simple folds too. Using μ to denote the least fixed point operator, the theorem is

$$(\!|R|\!) \cdot (\!|S|\!)^{\circ} \quad = \quad \mu X : R \cdot F\,X \cdot S^{\circ}$$

Unfortunately, hylomorphisms formed from generalised folds cannot be written as fixpoints, because the recursion is with a fold preceded by a map preceeded by an unfold, so there is no hylomorphism theorem for generalised folds. On the other hand, hylomorphisms that are formed from efficient folds are in a sense higher-order fixpoints, because the related operator χ_F is a fixpoint.

$$\chi_F\ (R'', S'') \quad = \quad \{\![R \mid R' \mid R'']\!\}_F \cdot \{\![S \mid S' \mid S'']\!\}_F^{\ \circ}$$

Unfortunately, to show that it is a least fixed point using the approach of [BdM97], we must not only adapt Knaster-Tarski to a lattice of operators like χ but also somehow lift left-division to the operators.

5.6 Fold operators preserve injectivity

We that know the efficient fold operator constructed in the endorelator category takes functions to functions. Now we shall show that it takes injective functions to injective functions, a result that will be needed in Chapter 8. First, however, we shall introduce the left-division operator, which we shall need in our proof.

5.6.1 Left-division

A *pre-specification* of R in S is an arrow X such that $R \cdot X \subseteq S$. The *weakest* pre-specification of R in S is the largest such X, and we shall write it as $R \backslash S$. In **Rel** we can read $\subseteq$ as refinement so $R \backslash S$ is an operation that when

performed before R establishes specification S. In particular, $R\backslash S$ denotes the largest such operation, that is, the one defined on the most inputs that has the most outputs for each input. We shall call the binary operator $\backslash$, *left-division*. (There is also a right-division operator but we will not need it.) The universal property for left-division is:

$$R \cdot X \subseteq S \;\; \equiv \;\; X \subseteq R\backslash S$$

The types are as follows. If $S : C \to B$ and $R : A \to B$ then $R\backslash S : C \to A$. If $R\backslash S$ is substituted for X then the right-to-left implication says that $R\backslash S$ is a pre-specification. The left-to-right implication says that $R\backslash S$ is the weakest (that is, the largest) pre-specification. Statements involving left-division often make more sense when turned into predicate logic, using

$$a\ (R\backslash S)\ c \;\; \equiv \;\; \forall b : \ b\, R\, a \Rightarrow b\, S\, c$$

We shall need to use a cancellation law in the proof:

$$R \cdot (R\backslash T) \subseteq T$$

We have defined left-division on relations, arrows of **Rel**, but we shall use it lifted to lax natural transformations, arrows of **Cor** (**Rel**). It is unclear whether left-division preserves lax natural transformations but our proofs will not require this property to hold.

5.6.2 Proof that fold operators preserve injectivity

A relation R is *injective* if $R^\circ \cdot R \subseteq id$. We shall show that $(\!|f \mid g|\!)_{NestF}$ is injective if f and g are injective, that is, the generalised fold operator for nests preserves injectivity. Then we shall generalise the proof to all nested datatypes.

$$
\begin{array}{ll}
 & (\!|f \mid g|\!)_{NestF}{}^\circ \cdot (\!|f \mid g|\!)_{NestF} \subseteq id \\
\equiv & \quad \{ \text{ definition of left-division } \} \\
 & (\!|f \mid g|\!)_{NestF} \subseteq (\!|f \mid g|\!)_{NestF}{}^\circ \backslash id \\
\Leftarrow & \quad \{ \text{ Knaster-Tarski } \} \\
 & f \cdot Base\ (id, ((\!|f \mid g|\!)_{NestF}{}^\circ \backslash id) \cdot Nest\, g) \cdot \alpha^\circ \subseteq (\!|f \mid g|\!)_{NestF}{}^\circ \backslash id \\
\equiv & \quad \{ \text{ definition of left-division } \} \\
 & (\!|f \mid g|\!)_{NestF}{}^\circ \cdot f \cdot Base\ (id, ((\!|f \mid g|\!)_{NestF}{}^\circ \backslash id) \cdot Nest\, g) \cdot \alpha^\circ \subseteq id
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \left\{ \begin{array}{l} \text{definition of generalised fold;} \\ \text{axioms of converse; } f \text{ is injective} \end{array} \right\} \\
& \alpha^\circ \cdot Base\ (id, Nest\ g^\circ \cdot (\!|f \mid g|\!)_{NestF}{}^\circ \cdot ((\!|f \mid g|\!)_{NestF}{}^\circ \backslash id) \cdot \\
& \quad Nest\ g) \cdot \alpha^\circ \subseteq id \\
\equiv & \{ \text{cancellation law; } Nest \text{ is a functor} \} \\
& \alpha^\circ \cdot Base\ (id, Nest\ (g^\circ \cdot g)) \cdot \alpha^\circ \subseteq id \\
\equiv & \{ g \text{ is injective; monotonicity of relators} \} \\
& \alpha \cdot \alpha^\circ \subseteq id \\
\equiv & \{ \alpha \text{ is an isomorphism} \} \\
& \text{true}
\end{array}
$$

The step that we need to generalise is the use of $g^\circ \cdot g \subseteq id$. For a hofunctor F_i with $F_i\, X = B_i \cdot \langle Id, X \cdot F_j\, X \rangle$

$$
\begin{array}{ll}
& (\Phi_i\ (\gamma, (\!|f \parallel \gamma|\!)_F))^\circ \cdot \Phi_i\ (\gamma, (\!|f \parallel \gamma|\!)_F{}^\circ \backslash id) \\
\subseteq & \{ \text{definition of } \Phi \} \\
& (\gamma_i \cdot B_i\ (id, (\!|f \parallel \gamma|\!)_F \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F))))^\circ \cdot \\
& \gamma_i \cdot B_i\ (id, ((\!|f \parallel \gamma|\!)_F{}^\circ \backslash id) \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F{}^\circ \backslash id))) \\
\subseteq & \left\{ \begin{array}{l} \text{axioms of converse; } \gamma_i \text{ is injective;} \\ B_i \text{ is a functor and relator} \end{array} \right\} \\
& B_i\ (id, T\ ((\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F))^\circ) \cdot (\!|f \parallel \gamma|\!)_F \\
& ((\!|f \parallel \gamma|\!)_F \backslash id) \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F{}^\circ \backslash id))) \\
\subseteq & \{ \text{cancellation law; monotonicity of relators} \} \\
& B_i\ (id, T\ ((\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F))^\circ) \cdot T\ (\Phi_j\ (\gamma, (\!|f \parallel \gamma|\!)_F{}^\circ \backslash id))) \\
\subseteq & \{ \text{induction hypothesis; monotonicity of relators} \} \\
& id
\end{array}
$$

It follows that the efficient fold operator also preserves injectivity.

$$
\begin{array}{ll}
& \{\![f \parallel \gamma \parallel h]\!\}^\circ \cdot \{\![f \parallel \gamma \parallel h]\!\} \\
= & \{ \text{specification of efficient folds} \} \\
& ((\!|f \parallel \gamma|\!)_F \cdot T\, h)^\circ \cdot (\!|f \parallel \gamma|\!)_F \cdot T\, h \\
\subseteq & \{ \text{axioms of converse; } T \text{ is a relator} \} \\
& T\, h^\circ \cdot (\!|f \parallel \gamma|\!)_F{}^\circ \cdot (\!|f \parallel \gamma|\!)_F \cdot T\, h
\end{array}
$$

$$
\begin{array}{ll}
\subseteq & \{ \text{ generalised fold operator preserves injectivity } \} \\
& T\, h^{\circ} \cdot T\, h \\
\subseteq & \{ \; h \text{ is injective; } T \text{ is a relator } \} \\
& id
\end{array}
$$

5.7 Fold-fusion law for efficient folds

We have already derived a fold-fusion law for efficient reductions on nests. Now we shall generalise the law to efficient folds on arbitrary datatypes.

$$
\begin{array}{ll}
& k \cdot \{\!\![f \parallel \gamma \parallel h]\!\!\}_F \\
= & \{ \text{ specification of efficient fold } \} \\
& k \cdot (\!| f \parallel \gamma |\!)_F \cdot T\, h \\
\subseteq & \left\{ \begin{array}{l} \text{fold-fusion:} \\ \textbf{assume } k \cdot f \subseteq f'' \cdot B\ (id, k \cdot R\, p_1) \\ \textbf{assume } M\, p_i \cdot \gamma_i \subseteq \gamma''_i \cdot B_i\ (id, k \cdot R\, p_j) \end{array} \right\} \\
& (\!| f'' \parallel \gamma'' |\!)_F \cdot T\, h \\
\subseteq & \left\{ \begin{array}{l} \text{map-fusion:} \\ \textbf{assume } f' \cdot B\ (k', id) \subseteq f'' \\ \textbf{assume } k' \cdot \gamma''_i \subseteq \gamma'_i \cdot B_i\ (k', id) \end{array} \right\} \\
& (\!| f' \parallel \gamma' |\!)_F \cdot T\, k' \cdot T\, h \\
\subseteq & \{ \text{ monotonicity of } T\text{; } \textbf{assume } k' \cdot h \subseteq h' \} \\
& (\!| f' \parallel \gamma' |\!)_F \cdot T\, h' \\
= & \{ \text{ specification of efficient fold } \} \\
& \{\!\![f' \parallel \gamma' \parallel h']\!\!\}_F
\end{array}
$$

Both conditions involving i are implicitly for all i. Also, i and j are related by $F_i\, X = B_i \cdot \langle Id, X \cdot F_j\, X \rangle$. The law that we have just derived, written below, will be specialised for efficient reductions and greatly simplified.

Fold-fusion law for efficient folds If we have

$$
\begin{aligned}
k \cdot f \subseteq f'' \cdot B\ (id, k \cdot R\, p_1) &\ \text{ and}\\
f' \cdot B\ (k', id) \subseteq f'' &\ \text{ and}\\
k' \cdot h \subseteq h'
\end{aligned}
$$

and for all i with $F_i\, X = B_i \cdot \langle Id, X \cdot F_j\, X\rangle$, we also have

$$
\begin{aligned}
M\, p_i \cdot \gamma_i \subseteq \gamma''_i \cdot B_i\ (id, k \cdot R\, p_j) &\ \text{ and}\\
k' \cdot \gamma''_i \subseteq \gamma'_i \cdot B_i\ (k', id)
\end{aligned}
$$

then we conclude that

$$k \cdot \{\![f \parallel \gamma \parallel h]\!\}_F \subseteq \{\![f' \parallel \gamma' \parallel h']\!\}_F$$

If M and R are constant functors then we can remove the p_i terms. We can in any case replace the inequalities by equalities. If we make both of these changes then we can remove f'' and γ'' to link f, f', γ and γ' directly. The first and second conditions follow from the single condition

$$k \cdot f = f' \cdot B\ (k', k)$$

If k is a function, then can derive a single condition that implies the third and fourth conditions.

$$
\begin{aligned}
& k' \cdot \gamma''_i = \gamma'_i \cdot B_i\ (k', id) \quad \text{and} \quad \gamma_i = \gamma''_i \cdot B_i\ (id, k)\\
\equiv \quad & \left\{ \text{ shunting function } k;\ B_i \text{ is a relator } \right\}\\
& k' \cdot \gamma''_i = \gamma'_i \cdot B_i\ (k', id) \quad \text{and} \quad \gamma''_i = \gamma_i \cdot B_i\ (id, k^\circ)\\
\Leftarrow \quad & \left\{ \text{ introducing } \gamma''_i \right\}\\
& k' \cdot \gamma_i \cdot B_i\ (id, k^\circ) = \gamma'_i \cdot B_i\ (k', id)\\
\equiv \quad & \left\{ \text{ shunting function } k;\ B_i \text{ is a relator and a functor } \right\}\\
& k' \cdot \gamma_i = \gamma'_i \cdot B_i\ (k', k)
\end{aligned}
$$

Now we have a much simpler fold-fusion law.

Generic fold-fusion law for efficient reductions If we have

$$\begin{aligned} k \cdot f &= f' \cdot B\ (k', k) \\ k' \cdot h &= h' \end{aligned}$$

for total function k and for all i we also have

$$k' \cdot \gamma_i \;=\; \gamma_i' \cdot B_i\ (k', k)$$

then we conclude

$$k \cdot \{\!\![f \parallel \gamma \parallel h]\!\!\}_F \;=\; \{\!\![f' \parallel \gamma' \parallel h']\!\!\}_F$$

Finally, as an aside, note that when $h = id$, we have a conclusion that connects map-fusion and fold-fusion on generalised folds.

$$k \cdot (\!|f \parallel \gamma|\!)_F \;=\; (\!|f' \parallel \gamma'|\!)_F \cdot T\ k'$$

Chapter 6

Membership

Now we shall start using the fold operators described in Chapter 3 and Chapter 4 and the relational framework described in Chapter 5 to define generic operations. The word generic is given an alternative very exclusive meaning in [Hoo97], on which much of this chapter and the next chapter is based, so when we wish to describe an operation as being parameterised by a functor, we shall use the different term, *polyfunctorial*. This chapter and the two that follow it will each define a polyfunctorial relation and show that it satisfies certain properties that constitute a specification. Although the three chapters refer to one another, they can be read separately in any order.

6.1 Introduction

Although we use relators to represent datatypes, not all relators correspond to datatypes. The only relators that do are those *with membership* [HdM00], that is, those having a membership relation with which stored values can be retrieved. (In [HdM00], the term container type is used to describe relators that have membership.) Hoogendijk showed in [Hoo97] that all regular relators have membership. In this chapter, we show that all nested relators have membership. To do this, we shall define a candidate membership relation for each nested relator and prove that it satisfies the characterisation of membership.

Formally, a *membership relation* for a relator F is a lax natural transformation $\eta : F \mathrel{\dot{\hookrightarrow}} Id$ with the interpretation that $a\,\eta_A\,x$ if and only if $a : A$

is an element of $x : F\,A$. In [Hoo97], Hoogendijk motivates a point-free property of membership relations and proves that it is a characterisation of membership. Consequently, every relator with membership has a unique membership relation and we will define in the next section, a polyfunctorial relation $\in$ that maps each nested relator T to its unique membership relation $\in_T$. This is only a candidate membership relation as we must check that $\in_T$ satisfies the characterisation of the membership relation for T. We can immediately guess what the membership relation for lists should be and we can write it pointwise with ellipses as follows.

$$a \in_{List} [a_0, \ldots, a_{n-1}] \quad \equiv \quad \exists i < n : \; a_i = a$$

We shall encounter, in the next section, a relator that does not have membership. If a relator F does have membership then it has a unique fan, denoted fan_F. The *fan* for a relator F is a lax natural transformation $\eta : Id \dot{\hookrightarrow} F$ with the interpretation that $x\,\eta_A\,a$ if and only if every element in x is equal to a. So the fan for lists non-deterministically maps a given seed a to a list of any length (including zero), all of whose elements are copies of a:

$$[a_0, \ldots, a_{n-1}] \; fan_{List} \; a \quad \equiv \quad \forall i < n : \; a_i = a$$

In fact we can define the fan of a relator F in terms of its membership relation as follows.

$$fan_F \quad = \quad \in_F \backslash id$$

It is clear from the definition of left-division that $x\,(\in_F \backslash id)\,a$ if x is an F-structure every member of which is equal to a.

Before continuing, we should note that other notions of datatypes in programming have been suggested, one of which is Jay's notion of shapely functors [Jay95]. It is shown in [HdM00] that being shapely is a sufficient condition for a functor both to be a relator and have membership.

Now we explain what is in the rest of this chapter. Section 6.2 gives the characterisation of membership. Section 6.3 defines a candidate membership relation for each nested relator. Section 6.4 shows that our candidates satisfy the characterisation by reducing the problem to that of showing that generalised unfans are efficient folds. This is proved by Section 6.6. To help with this, section 6.5 defines a polyfunctorial relation *fan* as the converse of an efficient fold.

6.2 Characterisation

Here is Oege de Moor's [HdM00] characterisation of membership:

Characterisation of membership A lax natural transformation η is a membership relation for an endorelator F, if for each $R : A \to B$ we have

$$F\,R \cdot\ (\eta_A \backslash id_A) \quad = \quad \eta_B \backslash R$$

To illustrate this characterisation, we confirm that the definition of $\in_{List}$ given in the introduction satisfies the equation above and is therefore the unique membership relation for lists. In other words, we demonstrate that

$$List\,R \cdot\ ((\in_{List})_A \backslash id) \quad = \quad (\in_{List})_B \backslash R$$

(In future, we shall drop such labelling of components of natural transformations.) The fan operation $\in_{List} \backslash id$ maps a value a to a list $[a_0, \ldots, a_{n-1}]$, for some $n \geq 0$, such that every a_i is equal to a. The map operation $List\,R$ then takes this list to the list $[b_0, \ldots, b_{n-1}]$ that satisfies $b_i\;R\;a$. By the definition of left-division, this left-hand side, which takes a to $[b_0, \ldots, b_{n-1}]$ is meant to be the largest relation such that $b \in b_i$ implies $b\;R\;a$. It clearly is the largest relation since there is no restriction on the length n of the list.

Hoogendijk motivates the characterisation from first principles without considering any particular datatype. That is why we shall regard it as a specification of membership and why we must prove that our candidate membership is correct with respect to it. The characterisation is easy to use in proofs, because it is non-inductive. For example, Hoogendijk and de Moor use it to show in [HdM00] that membership and fans are the largest lax natural transformations of their respective types. That is enough to define them uniquely. They also show that $\in_F \backslash \in_G$ is the largest natural transformation of type $G \dot{\hookrightarrow} F$. This result gives us an intuition for the expressivity of lax natural transformations: any element in the output structure must also have been in the input structure.

Finally, Freyd has located a relator that does not have membership. This is an important result because the notion of membership is far more useful if it actually excludes some relators. Consider the relator *Swap* with type $\mathbf{Rel} \times \mathbf{Rel} \to \mathbf{Rel} \times \mathbf{Rel}$ defined by $Swap\ (X, Y) = (Y, X)$. If *Swap* is

constructed starting with **Rel** as a base category then it does have membership but if it is constructed starting with $\mathbf{Rel} \times \mathbf{Rel}$ as a base category, so that *Swap* is in fact an endorelator on $\mathbf{Rel} \times \mathbf{Rel}$, then it does not have membership. This fact is proven in [Hoo97].

6.3 Candidate membership

The candidate membership $\in$ is defined by induction on the structure of nested relators. Hoogendijk constructs a regular fixpoint case from polynomial cases. We replace it with a more general nested fixpoint case constructed from the same polynomial cases. Then we confirm that the nested fixpoint case is a membership relation by adapting the corresponding proof that Hoogendijk gave for his regular fixpoint case.

6.3.1 Candidate membership: polynomial cases

As usual, we can easily guess the cases for polynomial endorelators from their types, which are

$$\begin{array}{rcl} (\in_{Id})_A & : & A \rightarrow A \\ (\in_{K_B})_A & : & B \rightarrow A \\ (\in_{F\times G})_A & : & F\,A + G\,A \rightarrow A \\ (\in_{F+G})_A & : & F\,A \times G\,A \rightarrow A \\ (\in_{F\cdot G})_A & : & F\;(G\,A) \rightarrow A \end{array}$$

The definitions, all of which Hoogendijk has confirmed, are

$$\begin{array}{rcl} \in_{Id} & = & id \\ \in_{K_B} & = & \emptyset_{-,B} \\ \in_{F\times G} & = & \in_F \cdot outl_{F,G} \cup \in_G \cdot outr_{F,G} \\ \in_{F+G} & = & \in_F \cdot {inl_{F,G}}^{\circ} \cup \in_G \cdot {inr_{F,G}}^{\circ} \\ \in_{F\cdot G} & = & \in_G \cdot (\in_F)_G \end{array}$$

Here, $\emptyset_{-,B}$ is defined by $(\emptyset_{-,B})_A = \emptyset_{A,B}$, where $\emptyset_{A,B} : A \rightarrow B$ denotes the empty relation from A to B.

In the product and coproduct cases, $\in_{F\times G}$ is short for $\in_{\times\cdot\langle F,G\rangle}$ and $\in_{F+G}$ is short for $\in_{+\cdot\langle F,G\rangle}$. In general for an arbitrary binary relator H,

$$\in_{H\cdot\langle F,G\rangle} \quad = \quad \in_F \cdot(\mathit{left}_H)_{\langle F,G\rangle} \cup \in_G \cdot(\mathit{right}_H)_{\langle F,G\rangle}$$

The functions $\mathit{left}_H : H\ (X,Y) \to X$ and $\mathit{right}_H : H\ (X,Y) \to Y$ will be defined shortly. For the product and coproduct cases, we have

$$\begin{aligned}\langle \mathit{left}_\times, \mathit{right}_\times\rangle &= \langle \mathit{outl}, \mathit{outr}\rangle\\ \langle \mathit{left}_+, \mathit{right}_+\rangle &= \langle \mathit{inl}^\circ, \mathit{inr}^\circ\rangle\end{aligned}$$

In general, the membership of a binary relator is a fork $\langle \mathit{left}_H, \mathit{right}_H\rangle$. Now, $(\mathit{left}_H)_{-,Y}$ is the membership relation of the endorelator $H\ (-,Y)$ obtained by right-sectioning H. Similarly, $(\mathit{right}_H)_{X,-}$ is the membership relation of the endorelator $H\ (X,-)$ obtained by left-sectioning H. So now we have

$$\begin{aligned}\in_\times &= \langle \mathit{outl}, \mathit{outr}\rangle\\ \in_+ &= \langle \mathit{inl}^\circ, \mathit{inr}^\circ\rangle\end{aligned}$$

For any $R : A \to B$ and $S : B \to C$, using the characterisation of membership, we have

$$\begin{aligned}H\ (R, \mathit{id}_Y)\cdot((\mathit{left}_H)_{A,Y}\backslash \mathit{id}_A) &= (\mathit{left}_H)_{C,Y}\backslash R\\ H\ (\mathit{id}_X, S)\cdot((\mathit{right}_H)_{X,B}\backslash \mathit{id}_B) &= (\mathit{right}_H)_{X,C}\backslash S\end{aligned}$$

The characterisation of membership for a binary relator H is, for $R : C \to A$ and $S : C \to B$

$$\begin{aligned}&H\ (R,S)\cdot\ ((\mathit{left}_H)_{C,C}\backslash \mathit{id}_C\ \cap\ (\mathit{right}_H)_{C,C}\backslash \mathit{id}_C)\\ &\qquad = ((\mathit{left}_H)_{A,B}\backslash R)\ \cap\ ((\mathit{right}_H)_{A,B}\backslash S)\end{aligned}$$

For example, for $F{=}\times$ this instantiates to

$$(R\times S)\cdot(\mathit{outl}_{C,C}\backslash \mathit{id}\ \cap\ \mathit{outr}_{C,C}\backslash \mathit{id}) \quad = \quad (\mathit{outr}_{A,B}\backslash R\ \cap\ \mathit{outr}_{A,B}\backslash S)$$

This is simply the absorption law of relational fork.

$$(R\times S)\cdot\langle \mathit{id}, \mathit{id}\rangle \quad = \quad \langle R, S\rangle$$

Finally, the candidate memberships for projections are as follows:

$$\begin{aligned}\in_{\mathit{Outl}} &= \langle \mathit{id}, \{\}\rangle\\ \in_{\mathit{Outr}} &= \langle \{\}, \mathit{id}\rangle\end{aligned}$$

6.3.2 Candidate membership: fixpoint case

We shall consider in this chapter an arbitrary nested relator T defined by

$$\begin{array}{rcl} T & \approx & F\,T \\ F\,X & = & B \cdot \langle Id, X \cdot F_1\,X \rangle \end{array}$$

There are an unspecified number of subsidiary hofunctors each of the form

$$F_i\,X \;=\; B_i \cdot \langle Id, X \cdot F_j\,X \rangle$$

It follows from the fact that α is a natural isomorphism that $\in_{F\,T} \cdot \alpha^\circ$ is a membership of $\in_T$: we argue

$$\begin{array}{ll} & (\in_{F\,T} \cdot \alpha^\circ)\backslash R \\ = & \quad \{ \text{ first law of division below } \} \\ & \alpha\backslash(\in_{F\,T} \backslash R) \\ = & \quad \{ \text{ second law of division below } \} \\ & \alpha \cdot (\in_{F\,T} \backslash R) \\ = & \quad \{ \in_{F\,T} \text{ is a membership } \} \\ & \alpha \cdot F\,T\,R \cdot (\in_{F\,T} \backslash id) \\ = & \quad \{ \alpha \text{ is a proper natural transformation } \} \\ & T\,R \cdot \alpha \cdot (\in_{F\,T} \backslash id) \\ = & \quad \{ \text{ second law of division below } \} \\ & T\,R \cdot \alpha^\circ\backslash(\in_{F\,T} \backslash id) \\ = & \quad \{ \text{ first law of division below } \} \\ & T\,R \cdot ((\in_{F\,T} \cdot \alpha^\circ)\backslash id) \end{array}$$

From this we conclude

$$\in_{F\,T} \;=\; \in_T \cdot \alpha$$

The laws of division that we used are respectively

$$\begin{array}{rcl} R\backslash(S\backslash T) & = & (S \cdot R)\backslash T \\ f^\circ \cdot X & = & f\backslash X \end{array}$$

The second law is restricted to functions f. We know that α° is a function because it is a natural isomorphism. We also know that α is a proper natural

transformation, a fact used elsewhere in the proof, for the same reason. If we substitute for F and use the polynomial cases of $\in$, we get the following

$$\begin{aligned} \in_{F\,T} &\;:\; F\,T \stackrel{.}{\hookrightarrow} Id \\ \in_{F\,T} &\;=\; (left_B)_{\langle Id,T\cdot F_1\,T\rangle} \cup \in_{F_1\,T} \cdot \in_T \cdot (right_B)_{\langle Id,T\cdot F_1\,T\rangle} \end{aligned}$$

The subscripts comes from the composition case. To help clarify the typing, observe

$$\begin{aligned} & (right_B)_{X,Y} \;:\; B\,(X,Y) \to Y \\ \equiv\;\; & right_B \;:\; B \stackrel{.}{\hookrightarrow} Outr \\ \Rightarrow\;\; & (right_B)_{\langle Id,T\cdot F_1\,T\rangle} \;:\; B \cdot \langle Id,\, T\cdot F_1\,T\rangle \stackrel{.}{\hookrightarrow} Outr \cdot \langle Id,\, T\cdot F_1\,T\rangle \\ \equiv\;\; & (right_B)_{\langle Id,T\cdot F_1\,T\rangle} \;:\; B \cdot \langle Id,\, T\cdot F_1\,T\rangle \stackrel{.}{\hookrightarrow} T\cdot F_1\,T \end{aligned}$$

Of course, $left_B$ is similar. Now we can equate the right-hand sides of the two equations for $\in_{F\,T}$ and shunt the function α to get

$$\in_T \;=\; ((left_B)_{\langle Id,T\cdot F_1\,T\rangle} \cup \in_{F_1\,T} \cdot \in_T \cdot (right_B)_{\langle Id,T\cdot F_1\,T\rangle}) \cdot \alpha^\circ$$

This equation has the form $X = \phi\, X$ where ϕ is a monotonic mapping, because composition and union are both monotonic in both arguments. By the Knaster-Tarski theorem, the equation has a solution. We shall show that any solution to this equation satisfies the characterisation of membership, but only one natural transformation can satisfy the characterisation of membership so the equation has exactly one solution.

Now we illustrate the definition of the candidate membership. Since composition distributes over union, $\in_{List}$ is given by

$$((left_{Base})_{\langle Id,List\rangle} \cdot \alpha^\circ) \;\cup\; (\in_{List} \cdot (right_{Base})_{\langle Id,List\rangle} \cdot \alpha^\circ)$$

Similarly, $\in_{Nest}$ is given by

$$((left_{Base})_{\langle Id,Nest\cdot Pair\rangle} \cdot \alpha^\circ) \;\cup\; (\in_{Pair} \cdot \in_{Nest} \cdot (right_{Base})_{\langle Id,Nest\cdot Pair\rangle} \cdot \alpha^\circ)$$

Finally, $\in_{Bush}$ is given by

$$((left_{Base})_{\langle Id,Bush\cdot Bush\rangle} \cdot \alpha^\circ) \;\cup\; (\in_{Bush} \cdot \in_{Bush} \cdot (right_{Base})_{\langle Id,Bush\cdot Bush\rangle} \cdot \alpha^\circ)$$

For regular relators, the candidate membership can be expressed more simply and with more insight. The head and the tail of a list are defined by

$$\begin{aligned} head &\;:\; List \stackrel{.}{\hookrightarrow} Id \\ head &\;=\; (left_{Base})_{\langle Id,List\rangle} \end{aligned}$$

$$tail \;:\; List \dot{\hookrightarrow} List$$
$$tail \;=\; (right_{Base})_{\langle Id, List\rangle}$$

So $\in_{List}$ can be rewritten as

$$\in_{List} \;=\; head \cup \in_{List} \cdot tail$$

Or using Kleene closure, we can even write

$$\in_{List} \;=\; head \cdot\; tail^{*}$$

This result states that we can get to any element of a list by repeatedly removing the first element any number of times (including zero) and taking the first element of whatever is left. We can write the membership of any other regular relator in a similar fashion.

6.4 Candidate membership is membership

We shall prove that for any nested relator G, our candidate membership for G satisfies the characterisation of membership, that is

$$G\, R \cdot (\in_G \backslash id) \;=\; \in_G \backslash R$$

Hoogendijk has already shown this for polynomial G so we only need to replace the regular fixpoint case of his proof with a nested fixpoint case. For arbitrary nested fixpoint T, we shall prove that

$$T\, R \cdot (\in_T \backslash id) \;=\; \in_T \backslash R$$

By applying converse to both sides, we see that this is equivalent to

$$(\in_T \backslash id)^{\circ} \cdot T\ (R^{\circ}) \;=\; (\in_T \backslash R)^{\circ}$$

We shall use the map-fusion law of efficient folds to prove this, after we have first written the expressions $(\in_T \backslash id)^{\circ}$ and $(\in_T \backslash R)^{\circ}$ as efficient folds. We choose efficient folds because simple folds have the wrong type and because it turns out that a proof with generalised folds gets stuck — we explain exactly how in Section 6.6. We plan to use a special case of the map-fusion law.

$$\{\![f \parallel \gamma \parallel id]\!\}_F \cdot T\ (R^{\circ}) \;=\; \{\![f \parallel \gamma \parallel R^{\circ}]\!\}_F$$

By comparing this with our requirement, we discover that we simply need to show, for some f and γ, that

$$(\in_T \backslash R)^\circ = \{\![f \parallel \gamma \parallel R^\circ]\!\}_F$$

We do this in Section 6.6 by using the universal property of efficient folds. To help us motivate our choice of f and γ, we now introduce the concept of unfans. The *unfan* for a relator G, denoted $unfan_G$, is the converse of the fan for G, that is

$$unfan_G = {fan_G}^\circ$$

Clearly, the unfan for G is the special case of $(\in_G \backslash R)^\circ$ with $R = id$, so we call $(\in_G \backslash R)^\circ$ a *generalised unfan* for G. To find out what f and γ might be, we just need to write $unfan_T$ as a reduction since

$$(\![f \parallel \gamma]\!)_F \;=\; \{\![f \parallel \gamma \parallel id]\!\}_F \;=\; (\in_T \backslash id)^\circ \;=\; unfan_T$$

This is what we shall do in the next section, where we shall also study fans at the same time. The approach we have just chosen differs from that of Hoogendijk [Hoo97] in two minor respects. First, instead of writing $(\in_T \backslash R)^\circ$ as a fold, he writes $\in_T \backslash R$ as an unfold. There does exist an efficient unfold operator, with its own universal property and fusion laws, but there seems little point in deriving them both just for this one proof. Secondly, we could try to derive f and γ in our proof like Hoogendijk does but we decide not to do this because the universal property of efficient folds is far more complex than that of standard unfolds.

6.5 Fans

6.5.1 Fans for polynomial cases

Hoogendijk defines [Hoo97] the following polynomial cases for the polyfunctorial relation *fan* and then verifies them by confirming for each polynomial relator F that fan_F equals $\in_F \backslash id$.

$$\begin{aligned} fan_{Id} &= id \\ fan_{Outl} &= id \\ fan_{Outr} &= id \end{aligned}$$

$$
\begin{array}{rcl}
fan_{K_A} & = & \Pi_{A,-} \\
fan_{+} & = & [id, id]^{\circ} \\
fan_{\times} & = & \langle id, id \rangle \\
fan_{F \cdot G} & = & F\ (fan_G) \cdot fan_F
\end{array}
$$

Here, $\Pi_{A,-}$ is defined by $(\Pi_{A,-})\, B = \Pi_{A,B}$, where $\Pi_{A,B}$ is the largest arrow of type $A \to B$ if it exists. For **Rel** and hence the endorelator category **Cor** (**Rel**), $\Pi_{A,B}$ is the product $A \times B$, which does exist for every A and B. As with the candidate membership, the unary cases can be guessed from their types, but the binary cases need further explanation. The fan for a binary relator B with membership $\langle left_B, right_B \rangle$ is given by

$$
fan_B \;=\; (left_B \backslash id) \;\cap\; (right_B \backslash id)
$$

We can calculate $fan_{\times}$ as follows.

$$
\begin{array}{ll}
 & (left_{\times} \backslash id) \;\cap\; (right_{\times} \backslash id) \\
= & \qquad \{ \text{ definition of } left \text{ and } right \text{ and candidate membership } \} \\
 & (outl \backslash id) \;\cap\; (outr \backslash id) \\
= & \qquad \{ \text{ law of left-division } \} \\
 & (outl^{\circ} \cdot id) \;\cap\; (outr^{\circ} \cdot id) \\
= & \qquad \{ \text{ definition of relational fork } \} \\
 & \langle id, id \rangle
\end{array}
$$

6.5.2 Unfans for nests

It is clear what an unfan should do. In pointwise form, $a\ unfan_F\ x$ is true exactly when every element in x is equal to a. In words, an unfan maps empty structures to every possible element and non-empty structures to a common element but the latter case is only defined when every element is the same. Suppose we write the unfan operation for nests as a reduction. Then the parameters have the types $Base\ (a, a) \to a$ and $Pair\ a \to a$, so we shall take them to be unfans also.

$$
\begin{array}{rcl}
unfan_{Nest} & = & Nest\ a \to a \\
unfan_{Nest} & = & (\!| unfan_{Base} \mid unfan_{Pair} |\!)_{NestF}
\end{array}
$$

This reduction replaces all pairs (x, x) with x. If it encounters a pair of different elements then the result of the whole reduction is undefined. Also *Base*-structures are treated in the same way so $unfan_{Nest}$ behaves as we wish it to.

6.5.3 Fans for fixpoint case

More generally, we have

$$fan_T \quad = \quad ((\![fan_B{}^{\circ} \,\|\, unfang]\!)_F)^{\circ}$$

Here, $unfang_i = fan_{B_i}{}^{\circ}$. If we wish, we can show that this candidate fan is equal to $\in_T \backslash id$ by proving that

$$(\in_T \backslash id)^{\circ} \quad = \quad \{\![fan_B{}^{\circ} \,|\, unfang \,|\, id]\!\}_F$$

The next section proves the generalisation of this where *id* is replaced by a relation R.

6.6 Generalised unfans are efficient folds

We begin by rehearsing our proof with the particular datatype of nests. This approach helped us when we showed that nested relators commute with converse, and that particular proof was also an application of the universal property of efficient folds. We shall need one more law of left-division, in addition to the two mentioned earlier in this chapter.

$$(R \,\cup\, S)\backslash T \quad = \quad (R\backslash T) \,\cap\, (S\backslash T)$$

We want to show that

$$(\in_{Nest} \backslash R)^{\circ} \quad = \quad \{\![fan_{Base}{}^{\circ} \,|\, fan_{Pair}{}^{\circ} \,|\, R^{\circ}]\!\}_{NestF}$$

If we let $\chi\,(R^{\circ})$ denote $(\in_{Nest} \backslash R)^{\circ}$ for all R, then this is

$$\chi\,(R^{\circ}) \quad = \quad \{\![fan_{Base}{}^{\circ} \,|\, fan_{Pair}{}^{\circ} \,|\, R^{\circ}]\!\}_{NestF}$$

Now the universal property gives us the following equivalent requirement:

$$\chi\,(R^{\circ}) \cdot \alpha \quad = \quad fan_{Base}{}^{\circ} \cdot Base\ (R^{\circ}, \chi\,(fan_{Pair}{}^{\circ} \cdot Pair\ R^{\circ}))$$

We break up the proof of this by proposing in our hints, two generic lemmas that will be proven later and reused. They fill in the gap created by working up from the bottom and down from the top.

$$
\begin{array}{ll}
& \chi\,(R^\circ) \cdot \alpha \\
= & \qquad \{ \text{ definition of } \chi \ \} \\
& (\in_{Nest} \backslash R)^\circ \cdot \alpha \\
= & \qquad \{ \text{ see first lemma below } \} \\
& (\in_{NestF\ Nest} \backslash R)^\circ \\
= & \qquad \{ \text{ see second lemma below } \} \\
& \mathit{fan_{Base}}^\circ \cdot \mathit{Base}\ (R^\circ, (\in_{Nest} \backslash (\in_{Pair} \backslash R))^\circ) \\
= & \qquad \{ \text{ definition of } \chi \ \} \\
& \mathit{fan_{Base}}^\circ \cdot \mathit{Base}\ (R^\circ, \chi\ ((\in_{Pair} \backslash R)^\circ)) \\
= & \qquad \{ \in_{Pair} \text{ is a membership } \} \\
& \mathit{fan_{Base}}^\circ \cdot \mathit{Base}\ (R^\circ, \chi\ ((\mathit{Pair}\, R \cdot (\in_{Pair} \backslash id))^\circ)) \\
= & \qquad \{ \text{ axioms of allegories; definition of } \mathit{fan}\text{; } \mathit{Pair} \text{ is a relator } \} \\
& \mathit{fan_{Base}}^\circ \cdot \mathit{Base}\ (R^\circ, \chi\,(\mathit{fan_{Pair}}^\circ \cdot \mathit{Pair}\ (R^\circ)))
\end{array}
$$

Observe that if we had tried to use generalised folds instead, we would get stuck in need of some way of introducing the functor *Nest*. Now we derive the two generic lemmas. The first is

$$(\in_T \backslash R)^\circ \cdot \alpha \quad = \quad (\in_{F\ T} \backslash R)^\circ$$

Its proof is

$$
\begin{array}{ll}
& (\in_T \backslash R)^\circ \cdot \alpha \\
= & \qquad \{ \text{ axioms of allegories } \} \\
& (\alpha^\circ \cdot (\in_T \backslash R))^\circ \\
= & \qquad \{ \text{ laws of division — } \alpha \text{ is a function } \} \\
& (\alpha \backslash (\in_T \backslash R))^\circ \\
= & \qquad \{ \text{ laws of division } \} \\
& ((\in_T \cdot \alpha) \backslash R)^\circ \\
= & \qquad \{ \text{ definition of candidate membership } \} \\
& (\in_{F\ T} \backslash R)^\circ
\end{array}
$$

To state the second generic lemma, let us generalise *NestF* to the hofunctor F_i defined by

$$F_i\, X = B_i \cdot \langle Id, X \cdot F_j\, X \rangle$$

Then we have

$$(\in_{F_i\, X} \backslash R)^\circ \;\; = \;\; \mathit{fan}_{B_i}{}^\circ \cdot B_i\ (R^\circ, (\in_X \backslash (\in_{F_j\, X} \backslash R))^\circ)$$

Our proof of this is

$$\begin{array}{ll}
 & (\in_{F_i\, X} \backslash R)^\circ \\
= & \quad \{ \text{ definition of } \in \ \} \\
 & ((\mathit{left}_{B_i} \cup \in_{F_j\, X} \cdot \in_X \cdot \mathit{right}_{B_i}) \backslash R)^\circ \\
= & \quad \{ \text{ laws of division; converse respects meet } \} \\
 & (\mathit{left}_{B_i} \backslash R)^\circ \ \cap \ (((\in_{F_j\, X} \cdot \in_X) \cdot \ \mathit{right}_{B_i}) \backslash R)^\circ \\
= & \quad \{ \text{ laws of division } \} \\
 & (\mathit{left}_{B_i} \backslash R)^\circ \ \cap \ (\mathit{right}_{B_i} \backslash ((\in_{F_j\, X} \cdot \in_X) \backslash R))^\circ \\
= & \quad \{ \text{ membership of binary relators; laws of division and converse } \} \\
 & ((\mathit{left}_{B_i} \backslash id)^\circ \ \cap \ (\mathit{right}_{B_i} \backslash id)^\circ) \cdot B_i\ (R^\circ, (\in_X \backslash (\in_{F_j\, X} \backslash R))^\circ) \\
= & \quad \{ \text{ fan of binary relators } \} \\
 & \mathit{fan}_{B_i}{}^\circ \cdot B_i\ (R^\circ, (\in_X \backslash (\in_{F_j\, X} \backslash R))^\circ)
\end{array}$$

Similarly, we also have

$$(\in_{F\, X} \backslash R)^\circ = \mathit{fan}_B{}^\circ \cdot B\ (R^\circ, (\in_X \backslash (\in_{F_1\, X} \backslash R))^\circ)$$

Now we can show the following in a generic fashion:

$$(\in_T \backslash R)^\circ \;\; = \;\; \{\![\mathit{fan}_B{}^\circ \parallel \mathit{unfang} \parallel R^\circ]\!\}_F$$

We let $\chi\ (R^\circ)$ be $(\in_T \backslash R)^\circ$ instead of $(\in_{Nest} \backslash R)^\circ$.

$$\chi\ (R^\circ) \;\; = \;\; \{\![\mathit{fan}_B{}^\circ \parallel \mathit{unfang} \parallel R^\circ]\!\}_F$$

The universal property says that this is equivalent to, for some ξ,

$$\begin{array}{rcl}
\chi\ (R^\circ) \cdot \alpha & = & \mathit{fan}_B{}^\circ \cdot B\ (R^\circ, \chi\ (\xi_1\ (R^\circ))) \\
\xi_i\ (R^\circ) & = & \mathit{fan}_{B_i}{}^\circ \cdot B_i\ (R^\circ, \chi\ (\xi_j\ (R^\circ)))
\end{array}$$

The second condition is for any hofunctor F_i defined as above. Now we have to think of a definition for ξ that will make both of these lines true. By analogy with our choice for χ, we try

$$\xi_i\ (R^\circ) = (\in_{F_i\ T} \backslash R)^\circ$$

We can immediately prove both conditions by using the two generic lemmas above.

$$\begin{array}{ll}
 & \xi_i\ R^\circ \\
= & \qquad \{\ \text{definition of } \xi\ \} \\
 & (\in_{F_i\ T} \backslash R)^\circ \\
= & \qquad \{\ \text{second lemma above}\ \} \\
 & \mathit{fan}_{B_i}{}^\circ \cdot B_i\ (R^\circ, (\in_T \backslash (\in_{F_j\ T} \backslash R))^\circ) \\
= & \qquad \{\ \text{definition of } \chi \text{ and } \xi\ \} \\
 & \mathit{fan}_{B_i}{}^\circ \cdot B_i\ (R^\circ, \chi\ (\xi_j\ (R^\circ)))
\end{array}$$

$$\begin{array}{ll}
 & \chi\ (R^\circ) \cdot \alpha \\
= & \qquad \{\ \text{definition of } \chi\ \} \\
 & (\in_T \backslash R)^\circ \cdot \alpha \\
= & \qquad \{\ \text{first lemma above}\ \} \\
 & (\in_{F\ T} \backslash R)^\circ \\
= & \qquad \{\ \text{second lemma above}\ \} \\
 & \mathit{fan}_B{}^\circ \cdot B\ (R^\circ, (\in_T \backslash (\in_{F_1\ T} \backslash R))^\circ) \\
= & \qquad \{\ \text{definition of } \chi \text{ and } \xi\ \} \\
 & \mathit{fan}_B{}^\circ \cdot B\ (R^\circ, \chi\ (\xi_1\ (R^\circ)))
\end{array}$$

Chapter 7

Zips

7.1 Introduction

In this chapter, as in the previous chapter, we use the foundation material of Chapters 3 and 5 to define a polyfunctorial operation and prove that it satisfies certain properties. More precisely, we shall define a candidate zip operation for pairs of linear nested relators by structural induction on the first argument, and then we shall show that the zip operation satisfies the properties expected of zips.

7.1.1 An example of a zip

The purpose of this chapter is to generalise the function *unzip*, below

$$\begin{array}{lcl} \mathit{unzip} & :: & [(a,b)] \rightarrow ([a],[b]) \\ \mathit{unzip} & = & \mathit{fork}\ (\mathit{map\,fst}, \mathit{map\,snd}) \end{array}$$

We can see that *unzip* takes a list of tuples to a tuple of lists. More abstractly, using the term F-*structure* to describe a value of type $x : F\ A$ for some A, we have that *unzip* takes a *List*-structure of $\times$-structures to a $\times$-structure of *List*-structures. The generalisation of *unzip* is a polyfunctorial relation *zip* that given two relators, F and G, turns an F-structure of G-structures into a G-structure of F-structures. If such a function exists, we say that F and G *commute*. When F and G are endorelators, the typing is

$$\mathit{zip}_{F,G} \ : \ F \cdot G \stackrel{.}{\rightarrow} G \cdot F$$

Recall that proper natural transformations like $zip_{F,G}$ can only shuffle elements around; they cannot duplicate or lose them. Now we can write *unzip* as an instance of *zip* but because $\times$ is a bifunctor, the type signature has a slightly different form.

$$\begin{array}{lcl} unzip & : & List \cdot \times \dot{\to} \times \cdot (List \times List) \\ unzip & = & zip_{List,\times} \end{array}$$

To see that this type is correct, observe that

$$\begin{array}{ll} & (List \cdot \times)\ (A, B) \dot{\to} (\times \cdot (List \times List))\ (A, B) \\ = & List\ (A \times B) \dot{\to} (List\ A) \times (List\ B) \end{array}$$

Note that the special case of *unzip* where the type variables are the same is $zip_{List,Pair}$.

7.1.2 Another example of a zip

The converse of *unzip* is *ununzip*, which has type $\times \cdot (List \times List) \dot{\to} List \cdot \times$. It is written in Haskell as

$$\begin{array}{lcl} ununzip & :: & ([a], [b]) \to [(a, b)] \\ ununzip\ ([\,], [\,]) & = & [\,] \\ ununzip\ (x : xs, y : ys) & = & (x, y) : ununzip\ (xs, ys) \end{array}$$

This is not the same as the function *zip* familiar to functional programmers because *zip* returns the empty list when one of its parameters has been used up. In contrast, *ununzip* is undefined in this situation, because it can only take pairs of equal length as *unzip* can only return pairs of equal length. Given the types of *zip* and *ununzip*, both of these functions are candidates for $zip_{\times,List}$. However, the specification of *ununzip* is easier to generalise to other datatypes than that of *zip* because it does not contain extra detail on how to handle pairs of lists of unequal length. Therefore, we decide

$$zip_{\times,List} \quad = \quad ununzip$$

Since we have argued that $zip_{List,\times}$ and $zip_{\times,List}$ should be mutual converses, we shall decide for any relators F and G, that $zip_{F,G}$ is the converse of $zip_{G,F}$. If this were not true, then polyfunctorial *zip* would superfluously define two different ways of turning $(F \cdot G)$-structures into $(G \cdot F)$-structures.

7.1.3 Informal specification

In order to specify *zip* fully, we must say when it is undefined. Therefore, we must generalise the notion of equal length of lists to datatypes other than lists. What notion should we use for trees? Equal depth, perhaps? Equal size would at least be applicable to lists as well, but we shall actually use a special case of equal size, namely equal shape. The *shape* of an F-structure $x : F\,A$ is given by $F\ !_A\, x$. This term is simply x with every element replaced by the unique value of the terminal object. Therefore, all pairs have equal shape and so do all lists of equal length. We call a value of type $F\,\mathbf{1}$, an F-*shape*. So an informal specification of *zip* is that $zip_{F,G}$ takes an F-structure of equally-shaped G-structures to a G-structure of equally shaped F-structures.

7.1.4 Applications

If matrices are represented by lists of lists then the transpose operation is $zip_{List,List}$. For example,

$$(zip_{List,List})_{Int}\ (\,[\,[1,2,3],[4,5,6]\,]\,)\quad=\quad(\,[\,[1,4],[2,5],[3,6]\,]\,)$$

We can also use *zip* to write a function of type $(A,[B]) \to [(A,B)]$, for any A and B, that broadcasts a value to each element of a list. An instance of *zip* having suitable type would be

$$(zip_{K_A\times Id,List})_B\quad:\quad((K_A\times Id)\cdot List)_B \stackrel{\cdot}{\to} (List\cdot(K_A\times Id))_B$$

Then an example would be

$$(zip_{K_{Char}\times Id,List})_{Int}\ ('a',[1,2,3])\quad=\quad[('a',1),('a',2),('a',3)]$$

This operation can, of course, be extended to datatypes other than lists, yielding a whole class of possible applications. Another application, that of structure multiplication, is explained in [HB97].

We finish with two uses that have been directly inspired by investigations into the area of nested datatypes. First, zips are used in one possible extension to nested datatypes of the substructures operation, a special case of the scan operation [BdMH96, BGB]. Secondly, Borges [Bor01] uses zips to write embedding functions for nested datatypes.

7.1.5 Multirelators

Hoogendijk defines zips for all regular relators and we shall generalise his work to linear nested relators. However, he extends the definition of regular relators to what he calls multirelators. These are relators whose source and target allegories both come from the closure of some base allegory **C**, which for us is **Rel**, under the construction of n-ary product allegories. The question then arises of how we can state and prove theorems about zips for every multirelator all at once. In particular, how can we compose arbitrary multirelators when writing the type of the zip ?

Hoogendijk's solution to this problem is to introduce the τ-Δ calculus. Unfortunately, this clutters his proofs to the point where they become difficult to read. Therefore, we shall mostly limit ourselves to endorelators in our proofs. However, we shall close the chapter by explaining how to adapt the proof requirements to arbitrary multirelators by introducing the τ-Δ calculus. No significant change in the proofs themselves are expected.

The definition of the candidate zip will be by induction (in the first argument) on the structure of nested relators. The grammars for polynomial hofunctors and our definition of nested functors are defined only for n-ary functors of the type $\mathbf{C}^n \to \mathbf{C}$. Consequently, we do not define a zip for $\mathit{Swap}{=}\langle \mathit{Outr}, \mathit{Outl}\rangle$ of type $\mathbf{Rel}^2 \to \mathbf{Rel}^2$. Hoogendijk, in contrast, does define zip for *Swap* because his class of regular relators is closed under the fork operation.

In our proofs, the only nested fixpoints we consider are endorelators. However, the polynomial relators from which they are built can be either unary or binary so we shall define zips for both of these cases. Finally, we shall assume that the second argument of the candidate zip is also an endorelator.

7.1.6 Chapter overview

Section 7.2 defines a candidate zip for every linear nested relator. Section 7.3 proposes some properties of zips and Section 7.4 proves that the candidate zip satisfies them. Section 7.5 explains how to adapt the proofs in the chapter to multirelators.

In order to make this thesis self-contained, we summarise much of Hoogendijk's work. It is hoped that this chapter will also act as an accessible introduction to zips. The paper [HB97] also serves this purpose but it has different emphasis. In particular, it highlights the connection between zips and strengths. Also, it clarifies the most challenging aspect of the thesis, the $\tau - \Delta$ calculus, whereas we have decided to try and do without it until the final section.

7.2 Inductive definition of candidate zip

We define *zip* by induction (in its first argument) on the structure of nested relators. First we give Hoogendijk's polynomial cases and then we give a nested fixpoint case, which is our replacement for his regular fixpoint case.

7.2.1 Candidate zips for polynomial endorelators

Recall that for any relators F and G, the type of $zip_{F,G}$ is $F \cdot G \dot{\to} G \cdot F$. However, we shall write zips as lax natural transformations, since we know they are arrows of the endorelator category, but we have not yet confirmed that they are also proper natural transformations.

$$\begin{array}{rcl} zip_{K_A,H} & : & A \to H\ A \\ zip_{Id,H} & : & H \dot{\hookrightarrow} H \\ zip_{F \cdot G,H} & : & F \cdot G \cdot H \dot{\hookrightarrow} H \cdot F \cdot G \\ zip_{G_1+G_2,H} & : & (G_1 \cdot H) + (G_2 \cdot H) \dot{\hookrightarrow} H \cdot (G_1 + G_2) \\ zip_{G_1 \times G_2,H} & : & (G_1 \cdot H) \times (G_2 \cdot H) \dot{\hookrightarrow} H \cdot (G_1 \times G_2) \end{array}$$

The last two cases are obtained by using the definitions of product and coproduct lifted to functors. Now we can easily suggest a definition for each of these cases based on their types. This motivation may seem a little long-winded (when we could have just copied the cases from Hoogendijk's thesis) but the manipulation we perform will make us familiar with those typing laws that must be understood if we are actually to use the candidate zip and understand its definition.

$$zip_{K_A,H} \quad = \quad (fan_H)_{K_A}$$

The identity case has the type of an identity natural transformation.

$$zip_{Id,H} = id_H$$

Since the definition of *zip* that we are headed for is inductive, we expect the composition case to be defined in terms of $zip_{F,H}$ and $zip_{G,H}$. Given that, the obvious way to construct a function of the required type is

$$zip_{F \cdot G,H} \quad = \quad (zip_{F,H})_G \cdot F\ zip_{G,H}$$

Recall that $(zip_{F,H})_G$ has the type $F \cdot H \cdot G \mathrel{\dot{\hookrightarrow}} H \cdot F \cdot G$.

The coproduct case has the type of a join of two natural transformations of types

$$G_1 \cdot H \mathrel{\dot{\hookrightarrow}} H \cdot (G_1 + G_2) \quad \text{and} \quad G_2 \cdot H \mathrel{\dot{\hookrightarrow}} H \cdot (G_1 + G_2)$$

Since these must use $zip_{G_1,H}$ and $zip_{G_2,H}$ in their definitions, we conclude

$$zip_{G_1+G_2,H} = [H\ inl_{G_1,G_2} \cdot zip_{G_1,H}, H\ inr_{G_1,G_2} \cdot zip_{G_2,H}]$$

The product case has the type of the converse of a fork of two natural transformations of types

$$H \cdot (G_1 \times G_2) \mathrel{\dot{\hookrightarrow}} G_1 \cdot H \quad \text{and} \quad H \cdot (G_1 \times G_2) \mathrel{\dot{\hookrightarrow}} G_2 \cdot H$$

These must also use $zip_{G_1,H}$ and $zip_{G_2,H}$ respectively in their definitions so we conclude

$$zip_{G_1 \times G_2,H} = \langle {zip_{G_1,H}}^{\circ} \cdot H\ outl_{G_1,G_2}, {zip_{G_2,H}}^{\circ} \cdot H\ outr_{G_1,G_2} \rangle^{\circ}$$

7.2.2 Candidate zips for polynomial binary relators

We must also consider cases where the first argument is a binary relator, since bifunctors appear in the grammar of nested endofunctors. However, we shall continue to assume that the second argument is an endorelator. We start with the projection relator *Outl* and work out the type of its zip as follows:

$$\begin{array}{ll} & zip_{Outl,H} : Outl \cdot (H \times H) \mathrel{\dot{\hookrightarrow}} H \cdot Outl \\ \Rightarrow & (zip_{Outl,H})_{(A,B)} : (Outl \cdot (H \times H))\ (A,B) \to (H \cdot Outl)\ (A,B) \\ \equiv & (zip_{Outl,H})_{(A,B)} : Outl\ (H\ A, H\ B) \to H\ (Outl\ (A,B)) \\ \equiv & (zip_{Outl,H})_{(A,B)} : H\ A \to H\ A \end{array}$$

So this case is a natural transformation η such that $\eta_{(A,B)} : H\ A \rightarrow H\ A$ and we choose the two projection cases to be

$$\begin{aligned} zip_{Outl,H} &= id_{H \cdot Outl} \\ zip_{Outr,H} &= id_{H \cdot Outr} \end{aligned}$$

The constant bifunctor case $zip_{KK_A,H}$ and the constant endofunctor case $zip_{K_A,H}$ have the same type so

$$zip_{KK_A,H} \quad = \quad (fan_H)_{K_A}$$

The composition case extends in the obvious way to

$$zip_{F \cdot \langle G_1, G_2 \rangle, H} \quad = \quad (zip_{F,H})_{\langle G_1, G_2 \rangle} \cdot F\ (zip_{G_1,H}, zip_{G_2,H})$$

Hoogendijk has a composition case only for $zip_{F \cdot G,H}$ but he can take G to be $\langle G_1, G_2 \rangle$, giving the equivalent definition

$$zip_{F \cdot \langle G_1, G_2 \rangle, H} \quad = \quad (zip_{F,H})_{\langle G_1, G_2 \rangle} \cdot F\ zip_{\langle G_1, G_2 \rangle, H}$$

Now we motivate a case for $zip_{\times,H}$.

$$\begin{array}{ll} & (zip_{\times,H})_{\langle G_1, G_2 \rangle} \cdot (zip_{G_1,H} \times zip_{G_2,H}) \\ = & \quad \{ \text{ definition of } zip \ \} \\ & zip_{\times \cdot \langle G_1, G_2 \rangle, H} \\ = & \quad \{ \text{ notation } \} \\ & zip_{G_1 \times G_2, H} \\ = & \quad \{ \text{ definition of } zip \ \} \\ & \langle zip_{G_1,H}{}^{\circ} \cdot H\ outl_{G_1,G_2}, zip_{G_2,H}{}^{\circ} \cdot H\ outr_{G_1,G_2} \rangle^{\circ} \\ = & \quad \{ \text{ absorption law } \} \\ & ((zip_{G_1,H}{}^{\circ} \times zip_{G_2,H}{}^{\circ}) \cdot \langle H\ outl_{G_1,G_2}, H\ outr_{G_1,G_2} \rangle)^{\circ} \\ = & \quad \{ \text{ axioms of converse } \} \\ & \langle H\ outl_{G_1,G_2}, H\ outr_{G_1,G_2} \rangle^{\circ} \cdot (zip_{G_1,H}{}^{\circ} \times zip_{G_2,H}{}^{\circ})^{\circ} \\ = & \quad \{ \text{ definition of lifted fork; } \times \text{ is a relator } \} \\ & (\langle H\ outl, H\ outr \rangle^{\circ})_{\langle G_1, G_2 \rangle} \cdot (zip_{G_1,H} \times zip_{G_2,H}) \end{array}$$

Hence, and after a similar argument for $+$, we conclude

$$\begin{aligned} zip_{\times,H} &= \langle H\ outl, H\ outr \rangle^{\circ} \\ zip_{+,H} &= [H\ inl, H\ inr] \end{aligned}$$

7.2.3 Candidate zips for nested fixpoints

Consider an arbitrary nested functor T.

$$\begin{array}{rcl} T & \approx & F\,T \\ F\,X & = & B \cdot \langle Id, X \cdot F_1\,X \rangle \end{array}$$

Hoogendijk's candidate zip is restricted to the case where $F_1\,X = Id$ so that T is regular. Rewriting his standard fold as a generalised fold gives us for regular T,

$$\begin{array}{rcl} zip_{T,H} & : & T \cdot H \dot{\hookrightarrow} H \cdot T \\ zip_{T,H} & = & (\!|H\,\alpha \cdot (zip_{B,H})_{\langle Id,T \rangle} \mid id_H |\!)_F \end{array}$$

Note that the type of $zip_{T,H}$ is not of the form $T \dot{\hookrightarrow} R$ so $zip_{T,H}$ cannot be a simple fold. Now we consider linear nested functors so let $F_1\,X{=}Q$ for some functor Q. Then the auxiliary parameter of the generalised fold must have the type $Q \cdot H \dot{\hookrightarrow} H \cdot Q$, which is the type of a zip and we conclude

$$zip_{T,H} \;=\; (\!|H\,\alpha \cdot (zip_{B,H})_{\langle Id,T\cdot Q \rangle} \mid zip_{Q,H} |\!)_F$$

We now show that the main parameter has the type expected. The second step is the absorption property of fork lifted to functors.

$$\begin{array}{ll} & zip_{B,H} \;:\; B \cdot (H \times H) \dot{\hookrightarrow} H \cdot B \\ \Rightarrow & \quad \{ \text{ typing rule of natural transformations } \} \\ & (zip_{B,H})_{\langle Id,T\cdot Q\rangle} \;:\; B \cdot (H \times H) \cdot \langle Id, T \cdot Q \rangle \dot{\hookrightarrow} H \cdot B \cdot \langle Id, T \cdot Q \rangle \\ \equiv & \quad \{ \text{ absorption property of fork lifted to functors } \} \\ & (zip_{B,H})_{\langle Id,T\cdot Q\rangle} \;:\; B \cdot \langle H, H \cdot T \cdot Q \rangle \dot{\hookrightarrow} H \cdot B \cdot \langle Id, T \cdot Q \rangle \\ \Rightarrow & \quad \{ \text{ definition of } T \text{ } \} \\ & H\,\alpha \cdot (zip_{B,H})_{\langle Id,T\cdot Q\rangle} \;:\; B \cdot \langle H, H \cdot T \cdot Q \rangle \dot{\hookrightarrow} H \cdot T \end{array}$$

Unfortunately, we do not know how to generalise *zip* further to non-linear datatypes. Define some hofunctor F_i used in the construction of F by, for some F_j

$$F_i\,X \;=\; B_i \cdot \langle Id, X \cdot F_j\,X \rangle$$

A generalised fold operator for T returns a result of type $T \cdot H \dot{\hookrightarrow} H \cdot T$ if it is given a parameter of type

$$B_i \cdot \langle H, H \cdot T \cdot F_j\;(H \cdot T) \rangle \dot{\hookrightarrow} H \cdot F_i\;(H \cdot T)$$

Applying the absorption law and substituting for F_i gives

$$B_i \cdot (H \times H) \cdot \langle Id, T \cdot F_j\ (H \cdot T)\rangle \stackrel{.}{\hookrightarrow} H \cdot B_i \cdot \langle Id, H \cdot T \cdot F_j\ (H \cdot T)\rangle$$

This does not have the type of $zip_{B_i,H}$, or even that of $(zip_{B_i,H})_G$ for some functor G, because an extra occurrence of H makes the two forks unequal. In fact, there probably is a natural transformation $T \cdot H \stackrel{.}{\hookrightarrow} H \cdot T$ for non-linear T that satisfies the properties of zips, but we do not know how to express it as a generalised fold so we cannot use the associated universal property and fusion laws to prove these properties.

7.2.4 Illustration of candidate zip

As an example, we shall implement $zip_{Nest,Tri}$, where *Tri* is some ternary relator, as a generalised fold on nests. By choosing *Tri* to be a ternary relator, we also demonstrate how easy it is to generalise the candidate zip beyond endorelators. We define *Pair* and *Tri* and *Base* in Haskell with

```
type Pair a     = (a, a)
data Tri a b c  = Tri a b c
data Base a b   = Nil | Cons (a, b)
```

First we implement $zip_{Pair,Tri}$ and $zip_{Base,Tri}$.

```
zipPT :: Pair (Tri a b c) → Tri (Pair a) (Pair b) (Pair c)
zipPT (Tri a b c, Tri d e f) = Tri (a, d) (b, e) (c, f)

zipBT :: B (Tri a b c) (Tri x y z) → Tri (B a x) (B b y) (B c z)
zipBT Nil = Tri Nil Nil Nil
zipBT (Cons (Tri a b c, Tri x y z)) =
          Tri (Cons (a, x)) (Cons (b, y)) (Cons (c, z))
```

We shall also need the map operations for the functors *Pair*, *Base* and *Tri*.

```
pair :: (a → b) → Pair a → Pair b
pair f (x, y) = (f x, f y)

tri :: (a → d) → (b → e) → (c → f) → Tri a b c → Tri d e f
tri f g h (Tri a b c) = Tri (f a) (g b) (h c)
```

$$
\begin{array}{l}
base :: (a \to c) \to (b \to d) \to Base\ a\ b \to Base\ c\ d \\
base\ f\ g\ Nil = Nil \\
base\ f\ g\ (Cons\ (x, y)) = Cons\ (f\ x, g\ y)
\end{array}
$$

Now we define *Nest* as follows.

$$
\begin{array}{l}
\textbf{type}\ NestF\ x\ a = Base\ a\ (x\ (Pair\ a)) \\
\textbf{newtype}\ Nest\ a = In\ (NestF\ Nest\ a)
\end{array}
$$

Following [BP99b], we define the generalised fold operator in a point-free style to make the link between the Haskell and the categorical definitions of $zip_{Nest,Tri}$ as clear as possible.

$$
\begin{array}{l}
out :: Nest\ a \to NestF\ Nest\ a \\
out\ (In\ x) = x
\end{array}
$$

$$
\begin{array}{l}
nest :: (a \to b) \to Nest\ a \to Nest\ b \\
nest\ f = In \cdot base\ f\ (nest\ (pair\ f)) \cdot out
\end{array}
$$

The type signature of *gfoldnest* will now be specialised to facilitate Haskell's first-order matching.

$$
\begin{array}{lll}
gfn & :: & \textbf{forall}\ m\ n\ r\ x\ y\ z. \\
& & (\textbf{forall}\ a\ b\ c. Base\ (m\ a\ b\ c)\ (n\ (r\ (Pair\ a))\ (r\ (Pair\ b)) \\
& & \qquad (r\ (Pair\ c))) \to n\ (r\ a)\ (r\ b)\ (r\ c)) \to \\
& & (\textbf{forall}\ a\ b\ c. Pair\ (m\ a\ b\ c) \to m\ (Pair\ a)\ (Pair\ b) \\
& & \qquad (Pair\ c)) \to \\
& & Nest\ (m\ x\ y\ z) \to n\ (r\ x)\ (r\ y)\ (r\ z) \\
gfn\ f\ g & = & f \cdot base\ id\ (gfn\ f\ g \cdot nest\ g) \cdot out
\end{array}
$$

Now we define the candidate zip as follows.

$$
\begin{array}{lll}
zipNT & :: & Nest\ (Tri\ a\ b\ c) \to Tri\ (Nest\ a)\ (Nest\ b)\ (Nest\ c) \\
zipNT & = & gfn\ ((tri\ In\ In\ In) \cdot zipBT)\ zipPT
\end{array}
$$

An example use is

$$
\begin{array}{ll}
& zipNT\ (In\ (Cons\ (Tri\ 1\,2\,3, In(Cons((Tri\ 4\,5\,6, Tri\ 7\,8\,9), In\ Nil))))) \\
= & Tri\ (In\ (Cons\ (1, In\ (Cons\ ((4,7), In\ Nil))))) \\
& \quad (In\ (Cons\ (2, In\ (Cons\ ((5,8), In\ Nil))))) \\
& \quad (In\ (Cons\ (3, In\ (Cons\ ((6,9), In\ Nil)))))
\end{array}
$$

Note that *zipNT* unzips a nest. Its converse zips nests together, but it is unclear what the use of such a function would be: it is partial so it is not always guaranteed to return a perfectly balanced tree. A similar function is given in Chapter 1.

7.3 Generic properties of zips

7.3.1 Formalising shape behaviour

To proof that $(\mathit{zip}_{F,G})_A$ has the desired shape behaviour for all A, we only need to check that it does for $A = \mathsf{1}$; as $\mathit{zip}_{F,G}$ is a natural transformation, we have

$$G\ (F\ !_A) \cdot (\mathit{zip}_{F,G})_A = (\mathit{zip}_{F,G})_{\mathsf{1}} \cdot F\ (G\ !_A)$$

An F-*shape* is an F-structure of $\mathsf{1}$'s. The operation $(\mathit{zip}_{F,G})_{\mathsf{1}}$ should take an F-structure of equal G-shapes to a G-structure of equal F-shapes. Since $\mathit{zip}_{F,G}$ and $\mathit{zip}_{G,F}$ are to be converses of each another, all we need is for the F-shapes in the target to be equal to the F-shape in the source. If x is an F-structure then $F\ (\mathit{fan}_G)_{\mathsf{1}}\ x$ is an F-structure with the same shape as x but containing G-shapes instead. Furthermore, the result of applying $(\mathit{zip}_{F,G})_{\mathsf{1}}$ to this is a G-structure of elements that all have the same shape as x and it is equal to $(\mathit{fan}_G)_{F\,\mathsf{1}}\ x$. We therefore require

$$(\mathit{fan}_G)_{\,F\,\mathsf{1}} \quad = \quad (\mathit{zip}_{F,G})_{\mathsf{1}} \cdot F\ (\mathit{fan}_G)_{\mathsf{1}}$$

7.3.2 Alternative requirements

We have stated precisely as a point-free equation, the shape requirement we introduced at the start. However, Hoogendijk found this operational requirement hard to prove directly so he proved that it followed from five less obvious requirements and then proved those instead. We have already explained two of these alternative requirements: that $\mathit{zip}_{F,G}$ should be a proper natural transformation and that it also be equal to the converse of $\mathit{zip}_{G,F}$. To explain the other requirements, let $\mathit{zip}_{F,-}$ be defined by $(\mathit{zip}_{F,-})_G = \mathit{zip}_{F,G}$. Hoogendijk requires that $\mathit{zip}_{F,-}$ be a homomorphism on the monoid of relators formed by the identity relator and the composition of relators.

$$\begin{aligned}(\mathit{zip}_{F,-})_{G\cdot H} &= (G\ (\mathit{zip}_{F,-})_H) \cdot (\mathit{zip}_{F,-})_G \\ (\mathit{zip}_{F,-})_{\mathit{Id}} &= \mathit{id}_F\end{aligned}$$

The final property, that zips be higher-order natural, requires rather more explanation.

7.3.3 Higher-order naturality

Reynolds [Rey83] proved an abstraction theorem for the polymorphic lambda calculus. It can be seen as a healthiness condition for any language like Haskell in which one can define polymorphic functions. Wadler popularised the theorem in [Wad89] as "Theorems for Free". He explained that every polymorphic function has a parametricity result that can be deduced from its type alone. A polymorphic function maps sets (or types) to functions. Interpreted in the category **Fun**, it maps an object A to an arrow of type $F\,A \to G\,A$ for functors F and G. Here F and G are functors. Wadler's claim is that all polymorphic functions are natural transformations of type $F \dot{\hookrightarrow} G$.

This claim does not hold for such object to arrow mappings in the endorelator category **Cor** (**Rel**). However, we motivate the final property of zips by finding what the naturality property would be if the claim held for one particular mapping: the polyfunctorial relation $zip_{F,-}$. Interpreted in the category **Cor** (**Rel**), a polyfunctorial relation maps an object H (a functor) to an arrow of type $\mathcal{F}\,H \to \mathcal{G}\,H$ (a lax natural transformation), where $\mathcal{F}$ and $\mathcal{G}$ are hofunctors, that is, endofunctors on **Cor** (**Rel**). In the case of $zip_{F,-}$, we have $\mathcal{F}{=}(F\cdot)$ and $\mathcal{G}{=}(\cdot F)$ and we shall demand that $zip_{F,-}$ is a natural transformation from $\mathcal{F}$ to $\mathcal{G}$.

$$
\begin{array}{llll}
(F\cdot)\,G & = F\cdot G & \qquad (\cdot F)\,G = G\cdot F \\
(F\cdot)\ \alpha & = F\,\alpha & \qquad (\cdot F)\ \alpha = \alpha_F
\end{array}
$$

To confirm the typing, observe that $zip_{F,G}$ can be written as a component of the natural transformation $zip_{F,-}$.

$$(zip_{F,-})_G \quad : \quad (F\cdot)\,G \dot{\hookrightarrow} (\cdot F)\,G$$

The naturality condition is said to be higher-order because it connects higher-order functors. For all $\beta : H \dot{\hookrightarrow} G$, we have that

$$(\cdot F)\ \beta\cdot(zip_{F,-})_H \quad = \quad (zip_{F,-})_G\cdot(F\cdot)\ \beta$$

Applying the definitions of $(F\cdot)$ and $(\cdot F)$, we get:

$$\beta_F\cdot zip_{F,H} \quad = \quad zip_{F,G}\cdot F\,\beta$$

A simple counterexample reveals why this equality must be weakened to an inequality. Suppose that $G{=}H{=}Pair$, with $F{=}List$, and $\beta = id_{Pair}\ \cup\ swap$

where $swap\ (x, y) = (y, x)$, so that β is the natural transformation that non-deterministically either swaps a pair or leaves it alone. First observe that swapping either all or none of the pairs in a list of pairs and then unzipping is the same as unzipping a list of pairs and non-deterministically either swapping the result or leaving it alone.

$$zip_{List,Pair} \ \cup \ (zip_{List,Pair} \cdot List\ swap) \quad = \quad (id \ \cup \ swap) \cdot zip_{List,Pair}$$

However, these relations are both more determined than the relation that swaps or leaves alone each of the pairs separately before unzipping.

$$(id \ \cup \ swap) \cdot zip_{List,Pair} \quad \subseteq \quad zip_{List,Pair} \cdot List\ (id \ \cup \ swap)$$

Putting these last two statements together and abstracting gives the *higher-order naturality property* of *zip*, which is that for all natural transformations $\beta : H \dot{\hookrightarrow} G$,

$$\beta_F \cdot zip_{F,H} \quad \subseteq \quad zip_{F,G} \cdot F\,\beta$$

A second reason for weakening the equality to an inequality is also given in Hoogendijk's thesis.

7.3.4 Hoogendijk's theorem of zips

Hoogendijk combines these requirements into a single theorem about the existence of zips. He proves the theorem by defining a candidate zip and showing that it has the required properties. We have just done the first of these so now we shall do the second. Since *zip* is defined by case analysis on the first argument, the properties in fact relate to the partial application of *zip*, that is $zip_{F,-}$ for some F.

Theorem of zips For all regular relators F, there is a polyfunctorial relation $zip_{F,-}$ that maps relators G that have membership to a collection of relations $zip_{F,G}$ indexed by objects and defined such that $(zip_{F,G})_A$ has type $F\ (G\ A) \to G\ (F\ A)$. In addition, $zip_{F,-}$ has four properties:

- $zip_{F,G} \ : \ F \cdot G \dot{\to} G \cdot F$
- $\beta_F \cdot zip_{F,H} \ \subseteq \ zip_{F,G} \cdot F\,\beta \qquad$ for all $\beta \ : \ H \dot{\hookrightarrow} G$
- $zip_{F,G \cdot H} \ = \ G\ zip_{F,H} \cdot zip_{F,G}$

- $zip_{F,Id} \; = \; id_F$

Finally, if G is also regular then $zip_{G,F} = zip_{F,G}{}^{\circ}$.

The left-hand sides of the third and fourth equations are both defined because the identity relator has membership and the composition of any two relators with membership also has membership. It is shown in [HB97] that zips are uniquely defined by these properties. The proof uses the universal property of fold operators.

7.4 Proofs of properties

Hoogendijk defines his candidate zip by structural induction in the first argument. In particular, he uses polynomial cases to define his regular fixpoint case. Unsurprisingly, his proof of the theorem is also by structural induction with a regular fixpoint case that assumes polynomial cases. We have replaced the regular fixpoint case of his definition with a nested fixpoint case of our own. Now the plan is to do the same in his proof.

He proves the first four properties separately and then uses them to prove the fifth. We would like to do exactly the same but there is a catch. In order to prove that zips are natural transformations we need the fifth property so we must prove that first. The other three properties are needed to do this so we will prove them even sooner. Crucially, the property that zips are natural transformations is not needed to prove the fifth property.

7.4.1 Zips are higher-order natural

We shall show that the higher-order naturality property of $zip_{T,H}$, for any H, follows from the higher-order naturality property of $zip_{B,H}$ and $zip_{Q,H}$. Hoogendijk has proven both of these properties since B and Q are polynomial, a fact we indicate with the hint "polynomial cases".

The fusion laws we shall use are the relational fusion laws specialised to linear nested datatypes.

$$(\!|f \mid g|\!)_F \cdot T\,k \;\supseteq\; (\!|f \cdot B\;(k, id) \mid g'|\!)_F \;\;\Leftarrow\;\; g \cdot Q\;k \;\supseteq\; k \cdot g'$$

$$k \cdot (\![f \mid g]\!)_F \;\subseteq\; (\![f' \mid g]\!)_F \quad\Leftarrow\quad k \cdot f \;\subseteq\; f' \cdot B\,(id, k)$$

Now here is our proof.

$$\begin{array}{ll}
& \beta_T \cdot zip_{T,H} \subseteq zip_{T,J} \cdot T\,\beta \\
\equiv & \qquad \{ \text{ definition of } zip \text{ } \} \\
& \beta_T \cdot (\![H\,\alpha \cdot (zip_{B,H})_{\langle Id, T\cdot Q\rangle} \mid zip_{Q,H}]\!)_F \\
& \qquad\qquad \subseteq (\![J\,\alpha \cdot (zip_{B,J})_{\langle Id, T\cdot Q\rangle} \mid zip_{Q,J}]\!)_F \cdot T\,\beta \\
\Leftarrow & \qquad \left\{ \begin{array}{l} \text{map-fusion: } \textbf{assuming } \beta_Q \cdot zip_{Q,H} \subseteq zip_{Q,J} \cdot Q\,\beta \\ \text{this is true by polynomial cases} \end{array} \right\} \\
& \beta_T \cdot (\![H\,\alpha \cdot (zip_{B,H})_{\langle Id, T\cdot Q\rangle} \mid zip_{Q,H}]\!)_F \\
& \qquad\qquad \subseteq (\![J\,\alpha \cdot (zip_{B,J})_{\langle Id, T\cdot Q\rangle} \cdot B\,(\beta, id) \mid zip_{Q,H}]\!)_F \\
\Leftarrow & \qquad \{ \text{ fold-fusion } \} \\
& \beta_T \cdot H\,\alpha \cdot (zip_{B,H})_{\langle Id, T\cdot Q\rangle} \\
& \qquad\qquad \subseteq J\,\alpha \cdot (zip_{B,J})_{\langle Id, T\cdot Q\rangle} \cdot B\,(\beta, id) \cdot B(id, \beta_T) \\
\Leftarrow & \qquad \{ \text{ naturality of } \beta : H \dot{\hookrightarrow} J;\ B \text{ is a functor } \} \\
& J\,\alpha \cdot \beta_T \cdot (zip_{B,H})_{\langle Id, T\cdot Q\rangle} \subseteq J\,\alpha \cdot (zip_{B,J})_{\langle Id, T\cdot Q\rangle} \cdot B\,(\beta, \beta_T) \\
\Leftarrow & \qquad \{ \text{ monotonicity of composition } \} \\
& \beta_T \cdot (zip_{B,H})_{\langle Id, T\cdot Q\rangle} \subseteq (zip_{B,J})_{\langle Id, T\cdot Q\rangle} \cdot B\,(\beta, \beta_T) \\
\Leftarrow & \qquad \{ \text{ polynomial cases } \} \\
& \text{true}
\end{array}$$

7.4.2 Zips are compositional

We wish to prove that

$$H\,zip_{T,I} \cdot (zip_{T,H})_I \quad=\quad zip_{T,H\cdot I}$$

To simplify the left-hand side and the proof itself, we introduce the abbreviation.

$$\eta_X \quad=\quad H\,zip_{X,I} \cdot (zip_{X,H})_I$$

Now we rewrite the right-hand side of our goal using the definition of *zip*.

$$\eta_T \quad=\quad (\![H\;(I\,\alpha) \cdot (zip_{B,H\cdot I})_{\langle Id, T\cdot Q\rangle} \mid zip_{Q,H\cdot I}]\!)_F$$

By the universal property of generalised folds,

$$\eta_T \cdot \alpha = H\,(I\,\alpha) \cdot (zip_{B,H \cdot I})_{\langle Id, T \cdot Q\rangle} \cdot B\ (id, \eta_T \cdot T\, zip_{Q,H \cdot I})$$

This is our new goal but first we prove a lemma. For any functor H,

$$zip_{T,H} \cdot (\alpha_F)_H \quad = \quad H\,\alpha_F \cdot zip_{B \cdot \langle Id, T \cdot Q\rangle, H}$$

To prove this we start with the definition of *zip*.

$$\begin{array}{ll} & zip_{T,H} = (\![H\,\alpha_F \cdot (zip_{B,H})_{\langle Id,T \cdot Q\rangle} \mid zip_{Q,H}]\!)_F \\ \equiv & \quad \{ \text{ universal property } \} \\ & zip_{T,H} \cdot (\alpha_F)_H = H\,\alpha_F \cdot (zip_{B,H})_{\langle Id,T \cdot Q\rangle} \cdot B\ (id, zip_{T \cdot H} \cdot T\ zip_{Q,H}) \\ \equiv & \quad \{ \text{ definition of } zip \ \} \\ & zip_{T,H} \cdot (\alpha_F)_H = H\,\alpha_F \cdot zip_{B \cdot \langle Id,T \cdot Q\rangle,H} \end{array}$$

Now we use the lemma to prove the goal.

$$\begin{array}{ll} & \eta_T \cdot \alpha_F \\ = & \quad \{ \text{ definition of } \eta \ \} \\ & H\,zip_{T,I} \cdot (zip_{T,H})_I \cdot (\alpha_F)_{H \cdot I} \\ = & \quad \{ \text{ lemma lifted with } I \ \} \\ & H\,zip_{T,I} \cdot H\,(\alpha_F)_I \cdot (zip_{B \cdot \langle Id,T \cdot Q\rangle,H})_I \\ = & \quad \{ \text{ lemma ; functor } H \ \} \\ & H\ (I\,\alpha_F) \cdot H\,zip_{B \cdot \langle Id,T \cdot Q\rangle,I} \cdot (zip_{B \cdot \langle Id,T \cdot Q\rangle,H})_I \\ = & \quad \{ \text{ definition of } \eta \ \} \\ & H\ (I\,\alpha_F) \cdot \eta_{B \cdot \langle Id,T \cdot Q\rangle} \\ = & \quad \{ \textbf{ claim}\text{: } \eta_{F \cdot G} = \eta_F \cdot F\,\eta_G \ \} \\ & H\ (I\,\alpha_F) \cdot (\eta_B)_{\langle Id,T \cdot Q\rangle} \cdot (B)(\eta_{\langle Id,T \cdot Q\rangle}) \\ = & \quad \{ \textbf{ claim}\text{: } \eta_{\langle F,G\rangle} = \langle \eta_F, \eta_G\rangle \ \} \\ & H\ (I\,\alpha_F) \cdot (\eta_B)_{\langle Id,T \cdot Q\rangle} \cdot (B)(\langle \eta_{Id}, \eta_{T \cdot Q}\rangle) \\ = & \quad \{ \text{ definition of fork; first claim } \} \\ & H\ (I\,\alpha_F) \cdot (\eta_B)_{\langle Id,T \cdot Q\rangle} \cdot B\,(\eta_{Id}, \eta_T \cdot T\,\eta_Q) \\ = & \quad \{ \text{ definition of } \eta\text{; polynomial cases } \} \\ & H\,(I\,\alpha_F) \cdot (zip_{B,H \cdot I})_{\langle Id,T \cdot Q\rangle} \cdot B\ (id, \eta_T \cdot T\,zip_{Q,H \cdot I}) \end{array}$$

Both claims are proven in Hoogendijk's thesis [Hoo97].

7.4.3 Zips respect identities

This property is the simplest of all to prove

$$
\begin{array}{ll}
 & zip_{T,Id} \\
= & \{ \text{ definition of } zip \text{ } \} \\
 & (\!|Id\,\alpha \cdot (zip_{B,Id})_{\langle Id, T\cdot Q\rangle} \mid zip_{Q,Id}|\!) \\
= & \{ \text{ polynomial cases } \} \\
 & (\!|\alpha \mid id|\!)_F \\
= & \{ \text{ } id_T \cdot \alpha = \alpha \cdot B\ (id_{Id}, id_T \cdot T\ id_Q)\text{; universal property } \} \\
 & id_T
\end{array}
$$

7.4.4 Linear nested relators are almost commuting

We need to show that for linear nested relators T and U,

$$zip_{T,U} \quad = \quad zip_{U,T}{}^{\circ}$$

By the definition of *zip* and the universal property of generalised folds, this is equivalent to

$$zip_{U,T}{}^{\circ} \cdot \alpha_F = (U\,\alpha_F \cdot (zip_{B,U})_{\langle Id, T\cdot Q\rangle}) \cdot B\ (id, zip_{U,T}{}^{\circ} \cdot T\ zip_{Q,U})$$

Before proving this, we show that we can rewrite the left-hand side.

$$
\begin{array}{ll}
 & zip_{U,T}{}^{\circ} \cdot (\alpha_F)_U = U\,\alpha_F \cdot zip_{U,F\,T}{}^{\circ} \\
\equiv & \{ \text{ converse contravariant and order-preserving; } U \text{ relator } \} \\
 & ((\alpha_F)_U)^{\circ} \cdot zip_{U,T} = zip_{U,F\,T} \cdot U\,\alpha_F{}^{\circ} \\
\Leftarrow & \{ \text{ antisymmetry of } \subseteq \text{ } \} \\
 & ((\alpha_F)_U)^{\circ} \cdot zip_{U,T} \subseteq zip_{U,F\,T} \cdot U\,\alpha_F{}^{\circ} \\
 & ((\alpha_F)_U)^{\circ} \cdot zip_{U,T} \supseteq zip_{U,F\,T} \cdot U\,\alpha_F{}^{\circ} \\
\equiv & \{ \text{ } \alpha \text{ is a function and therefore can be shunted } \} \\
 & ((\alpha_F)_U)^{\circ} \cdot zip_{U,T} \subseteq zip_{U,F\,T} \cdot U\,\alpha_F{}^{\circ} \\
 & ((\alpha_F)_U) \cdot zip_{U,T} \subseteq zip_{U,F\,T} \cdot U\alpha_F \\
\equiv & \left\{ \begin{array}{l} \text{zips are higher-order natural} \\ \text{naturality of } \alpha_F : F\ T \stackrel{\cdot}{\hookrightarrow} T \text{ and } \alpha_F{}^{\circ} : T \stackrel{\cdot}{\hookrightarrow} F\ T \end{array} \right\} \\
 & \text{true}
\end{array}
$$

Now we reason

$$
\begin{array}{ll}
 & U\ \alpha_F \cdot {zip_{U,F\ T}}^{\circ} \\
= & \quad \{\ \text{definition of hofunctor } F\ \} \\
 & U\ \alpha_F \cdot {zip_{U,B\cdot\langle Id,T\cdot Q\rangle}}^{\circ} \\
= & \quad \{\ \textbf{claim}\text{: } {zip_{U,F\cdot G}}^{\circ} = ({zip_{U,F}}^{\circ})_G \cdot F\ {zip_{U,G}}^{\circ}\ \} \\
 & U\ \alpha_F \cdot ({zip_{U,B}}^{\circ})_{\langle Id,T\cdot Q\rangle} \cdot (B)({zip_{\langle Id,T\cdot Q\rangle}}^{\circ}) \\
= & \quad \{\ \textbf{claim}\text{: } {zip_{U,\langle F,G\rangle}}^{\circ} = \langle {zip_{U,F}}^{\circ}, {zip_{U,G}}^{\circ}\rangle\ \} \\
 & U\ \alpha_F \cdot ({zip_{U,B}}^{\circ})_{\langle Id,T\cdot Q\rangle} \cdot B\ ({zip_{U,Id}}^{\circ}, ({zip_{U,T\cdot Q}}^{\circ})_Q) \\
= & \quad \{\ \text{first claim; zips respect identities}\ \} \\
 & U\ \alpha_F \cdot ({zip_{U,B}}^{\circ})_{\langle Id,T\cdot Q\rangle} \cdot B\ (id, ({zip_{U,T}}^{\circ})_Q \cdot T\ {zip_{U,Q}}^{\circ}) \\
= & \quad \{\ \text{polynomial cases}\ \} \\
 & (U\ \alpha_F \cdot (zip_{B,U})_{\langle Id,T\cdot Q\rangle}) \cdot B\ (id, ({zip_{U,T}}^{\circ})_Q \cdot T\ zip_{Q,U})
\end{array}
$$

Applying converse to both sides of our two claims leaves

$$
\begin{array}{r}
zip_{U,F\cdot G} = F\ zip_{U,G} \cdot (zip_{U,F})_G \\
zip_{U,\langle F,G\rangle} = \langle zip_{U,F}, zip_{U,G}\rangle
\end{array}
$$

The first of these asserts that zips are compositional, something we have just proved. The second is proved by Hoogendijk in his thesis.

7.4.5 Zips are natural transformations

We did not use in the previous proof the fact that zips are natural transformation so we can now show this using the result shown by the previous proof.

We show, for all $R : A \to B$, that

$$G\ (T\ R) \cdot (zip_{T,G})_A \quad = \quad (zip_{T,G})_B \cdot T\ (G\ R)$$

This time, we cannot adapt the proof given by Hoogendijk for regular relators. For regular T, he uses the map-fusion law to rewrite the right-hand side to a fold. Then he applies the fold-fusion law and gets conditions that follow from the naturality of $zip_{B,G}$. Unlike its counterpart for standard folds, the map-fusion law for generalised folds has a condition: we must solve for g in

$$zip_{Q,G} \cdot Q\ (G\ R) \quad = \quad (G\ R)_Q \cdot g$$

The type of g is $Q \cdot G \cdot K_A \dot{\to} G \cdot K_B \cdot Q$. Suppose that $Q = Pair$, that $G = Id$ and that $A = B$. Then g has the type $Pair\, A \to A$. However, if g is $outl_{A,A}$ or $outr_{A,A}$ then the equation cannot hold, since one side would remove elements while the other would not. The fold-fusion law of efficient folds is not of any use either for the same reason.

So we must take a different approach. Instead of using the definition of $zip_{T,G}$ directly, we shall use merely the fact that $zip_{T,G}$ is a generalised fold and hence a lax natural transformation. To strengthen this to a proper natural transformation, we must show that

$$G\ (T\ R) \cdot (zip_{T,G})_A \ \subseteq\ (zip_{T,G})_B \cdot T\ (G\ R)$$

Axioms of converse and the fact that G and T are relators make this equivalent to

$$(zip_{T,G})_A{}^{\circ} \cdot G\ (T\ (R^{\circ})) \ \subseteq\ T\ (G\ (R^{\circ})) \cdot (zip_{T,G})_B{}^{\circ}$$

Since linear nested relators commute, this is equivalent to

$$T\ (G\ (R^{\circ})) \cdot (zip_{G,T})_B \ \supseteq\ (zip_{G,T})_A \cdot G\ (T\ (R^{\circ}))$$

If G is polynomial then Hoogendijk has shown that $zip_{G,T}$ is a proper natural transformation and is therefore lax too. On the other hand, if G is the fixpoint case then $zip_{G,T}$ is still a lax natural transformation because it is a generalised fold.

This completes the extension of Hoogendijk's theorem to linear nested relators.

7.5 From endorelators to multirelators

We conclude this chapter by showing how to adapt the properties of zips (and hence their proofs) to multirelators. In general, for any category $\mathbf{C}$ if relator F has type $\mathbf{C}^k \to \mathbf{C}^m$ and relator G has type $\mathbf{C}^l \to \mathbf{C}^n$, then the type of $zip_{F,G}$ is written in the notation of the τ-Δ calculus as

$$zip_{F,G} \ :\ ({}^nF)\,(G^k) \dot{\to} (G^m)({}^lF)$$

These subscripts are explained by the following rules.

$$F : \mathbf{C}^m \to \mathbf{C}^n \;\Rightarrow\; F^k : (\mathbf{C}^m)^k \to (\mathbf{C}^n)^k$$
$$F : \mathbf{C}^m \to \mathbf{C}^n \;\Rightarrow\; {}^kF : (\mathbf{C}^k)^m \to (\mathbf{C}^k)^n$$

So we have, for example,

$$({}^nF)\,(G^k) \;:\; (\mathbf{C}^l)^k \to (\mathbf{C}^n)^m$$
$$(G^m)\,({}^lF) \;:\; (\mathbf{C}^l)^k \to (\mathbf{C}^n)^m$$

Indeed the subscripts can be lifted to natural transformations with the rules.

$$\alpha : F \dot{\hookrightarrow} G \;\Rightarrow\; \alpha^k : F^k \dot{\hookrightarrow} G^k$$
$$\alpha : F \dot{\hookrightarrow} G \;\Rightarrow\; {}^k\alpha : {}^kF \dot{\hookrightarrow} {}^kG$$

We have already stated that $zip_{F,G}$ must be a proper natural transformation. The higher-order naturality property is that for any $G, H : \mathbf{C}^l \to \mathbf{C}^n$ and $\alpha : G \dot{\hookrightarrow} H$,

$$(\alpha^m)_{({}^lF)} \cdot zip_{F,G} \;\subseteq\; zip_{F,H} \cdot ({}^nF)\,(\alpha^k)$$

The property that zips respect compositions is for $G : \mathbf{C}^l \to \mathbf{C}^n$ and $H : \mathbf{C}^o \to \mathbf{C}^l$,

$$zip_{F,G\cdot H} \;=\; (G^m)\,(zip_{F,H}) \cdot (zip_{F,G})_{H^k}$$

Instead of requiring that zip respects identities, we require that they respect projections.

$$zip_{F,\Pi^l_i} \;=\; id_{F\,Proj^k}$$

The new candidate zip is

$$zip_{T,G} \;=\; (\!|G\,{}^l\alpha \cdot (zip_{B,G})_{{}^l\langle Id,T\rangle}|\!)_F$$

Here, the functor G has type $\mathbf{C}^l \to \mathbf{C}$. A theorem proved in [Hoo97] extends the candidate zip to more general G. An instance of the theorem that illustrates the idea is

$$zip_{F,\langle G_1,G_2\rangle} \;=\; \tau\,\langle zip_{F,G_1}, zip_{F,G_2}\rangle$$

Here, τ is a transposition functor of type $(\mathbf{C}^{k'})^{l'} \to (\mathbf{C}^{l'})^{k'}$ defined for any k' and l'. We also use τ to generalise the statement that zips are commuting.

Chapter 8

Embedding Functions

8.1 Introduction

As we explained in Chapter 1, nested datatypes do not enhance the expressivity of Haskell. When a nested datatype is used in a program, it can always be replaced by the composition of two regular datatypes. We call this substitute an *embedding target*. Echoing the slogan of Chapter 1, we can say that the embedding target is a "nested datatype minus constraints".

In this chapter, we shall define for every nested datatype a regular embedding target and an associated embedding function that maps values of the nested datatype to values of the embedding target. We can use the embedding function to define an equivalence between programs with properly nested datatypes and programs without, and to deduce one from the other. We shall also define for one particular datatype an embedding predicate that tests whether a value of the embedding target is in the range of the embedding function.

As an example, let us find an embedding target for the functor *Nest*. First, let the term *k-fold pair* of *a*'s describe a value of type $Pair^k\ a$. Then a nest is a 0-fold pair followed by a 1-fold pair followed by a 2-fold pair and so on. Each k-fold pair is a perfect leaf-labelled tree of height k written with no constructors. A nest can consequently be represented by a list of leaf-labelled trees so an embedding target for *Nest* is *NestR*, defined by

$$NestR \quad = \quad List \cdot Tree$$

$$
\begin{array}{lcl}
\mathit{List}\ A & \approx & 1 + A \times \mathit{List}\ A \\
\mathit{Tree}\ A & \approx & A + \mathit{Pair}\ (\mathit{Tree}\ A)
\end{array}
$$

More formally, there is a functor $NestR'$ such that for any set A, $NestR'\ A$ is both isomorphic to $Nest\ A$ and a subset of $NestR\ A$ using the naive notion of "types as sets". To demonstrate this, we define an injective function, called an *embedding function*, from *Nest* to *NestR*.

$$
\mathit{embednest} \quad : \quad \mathit{Nest} \dot{\to} \mathit{NestR}
$$

Section 8.2 discusses further the choice of *NestR* as embedding target and Section 8.3 defines and verifies *embednest*.

Now *embednest* can be generalised to a polyfunctorial relation *embed* such that $embed_T$ has the type $T \dot{\to} \overline{T}$. Here, $\overline{T}$ denotes the embedding target of T, so $\overline{Nest}=NestR$, for example. Section 8.4 defines $\overline{T}$ and $embed_T$ for the case where $T = Bush$. Section 8.5 defines $\overline{T}$ and $embed_T$ for the case where T is any properly nested relator.

More abstractly, *embed* connects the class of nested relators with the class of regular relators, enabling us to write equations that connect programs that use nested datatypes with programs that use only regular datatypes. Let f have type $X \dot{\to} Y$, where X and Y are nested relators. Then there is a corresponding function between regular relators $f' : X' \dot{\to} Y'$, given by

$$
embed_Y \cdot f \quad = \quad f' \cdot embed_X
$$

Naturally, for regular relators the embedding function is the identity. Section 8.7 proves a theorem, known as the fold-equivalence law, that gives f' in the case where f is an efficient reduction and Y is regular. In order to state and prove this theorem, Section 8.6 gives a fold-fusion law for a variant of the efficient reduction operator. Section 8.8 considers arbitrary recursion and Section 8.9 proves a variant of the fold-equivalence law where Y need not be regular but it applies only to so-called tail-recursive datatypes. Finally, Section 8.10 uses the existence of embedding functions to give an alternative proof that nested relators have membership and Section 8.11 derives an embedding predicate for the type of non-empty nests.

Although this chapter has nothing to say about the relative efficiencies of

f and f', Okasaki's thesis [Oka98b] has a comparison for basic operations on the datatype of random-access lists. In general, nested datatypes are sometimes slightly more efficient because there are fewer constructor functions to unpack. However, there are other reasons for wanting to calculate f from f' or vice versa. Haskell functions for properly nested datatypes will feature polymorphic recursion in general, which is banned by some languages, and their folds are less well understood than standard folds. Therefore, we may want to remove nested datatypes from our programs. On the other hand, our laws can be used instead to construct programs for nested datatypes from simpler programs for regular datatypes.

8.2 Embedding target for nests

Here is another way of motivating *NestR*, one that can easily be adapted to other linear nested datatypes. Let $X \lessapprox Y$ mean that X is isomorphic to a subset of Y. We simply expand the definition of *Nest* a few times and look for patterns, guided by an assumption that we are heading for a composition of datatypes.

$$
\begin{array}{ll}
 & \mathit{Nest}\ A \\
\approx & \quad \{\ \text{definition of } \mathit{Nest} \text{ (three times)}\ \} \\
 & \mathit{Base}\ (A, \mathit{Base}\ (\mathit{Pair}\ A, \mathit{Base}\ (\mathit{Pair}^2\ A, \mathit{Nest}\ (\mathit{Pair}^3\ A)))) \\
\lessapprox & \quad \{\ \text{assumption}\ \} \\
 & \mathit{Base}\ (A, \mathit{Base}\ (\mathit{Pair}\ A, \mathit{Base}\ (\mathit{Pair}^2\ A, U\ (V\ (\mathit{Pair}^3\ A))))) \\
\lessapprox & \quad \{\ \mathbf{let}\ V\ A \approx A + \mathit{Pair}\ (V\ A)\ \} \\
 & \mathit{Base}\ (V\ A, \mathit{Base}\ (V\ A, \mathit{Base}\ (V\ A, U\ (V\ A)))) \\
\approx & \quad \{\ \mathbf{let}\ U\ A \approx \mathit{Base}\ (A, U\ A)\ \} \\
 & U\ (V\ A)
\end{array}
$$

So *Nest* does indeed have $\mathit{List} \cdot \mathit{Tree}$ as one of its embedding targets. Here, *List* and *Tree* are defined in the notation of category theory by

$$
\begin{array}{rcl}
\mathit{List} & \approx & \mathit{ListF}\ \mathit{List} \\
\mathit{ListF}\ X & = & \mathit{Base} \cdot \langle \mathit{Id}, X \rangle \\
 & & \\
\mathit{Tree} & \approx & \mathit{TreeF}\ \mathit{Tree}
\end{array}
$$

$$\begin{aligned} \mathit{TreeF}\ X &= \mathit{Base}' \cdot \langle \mathit{Id}, X \rangle \\ \mathit{Base}'\ (X, Y) &= X + \mathit{Pair}\ Y \end{aligned}$$

However, we noted in Chapter 1 that each nest corresponds to the levels of a complete internally-labelled binary tree.

$$\mathit{ITree}\ A \approx \mathsf{1} + A \times \mathit{Pair}\ \ (\mathit{ITree}\ A)$$

Borges [Bor01] uses this simpler datatype as the embedding target for *Nest*. The embedding function is easy to write if $\mathit{zip}_{\mathit{ITree},\mathit{Pair}}$ is supplied and the existence of this zip both motivates the embedding target and justifies the embedding function.

$$\begin{aligned} \mathit{zembednest} &: \mathit{Nest} \dot{\to} \mathit{ITree} \\ \mathit{zembednest} &= (\![\alpha \cdot (\mathit{id} + \mathit{id} \times \mathit{zip}_{\mathit{ITree},\mathit{Pair}})]\!)_{\mathit{NestF}} \end{aligned}$$

However, there are many drawbacks to using *ITree*:

- the embedding function is inefficient because it repeatedly unzips its input.
- the unzipping also changes the syntactic order of the elements
- the approach does not extend to non-linear datatypes
- in any case, initial algebras are more fundamental than zips
- *ITree* is less closely linked to *Nest*. *ITree* is better suited to depth-first traversals, whereas *Nest* and *NestR* are better suited to breadth-first traversals as they both reveal their nodes one level at a time.

There is, in fact, another nested datatype for internally-labelled binary trees that like *Nest* is more suited to depth-first traversals. It is a slimmed down version of the datatype for Braun trees and is used in [BGB] to implement pattern-matching for perfect trees.

$$\mathbf{data}\ \mathit{NestIter}\ a\ b = \mathit{Zero}\ b \mid \mathit{Succ}\ (\mathit{NestIter}\ a\ (b,\ a,\ b))$$

Finally, we should note that flattening a nest is injective so we could use *eflatnest* as an embedding function for nests. However, flattening is not injective for all nested datatypes so we cannot use it as an embedding function in general.

8.3 Embedding function for nests

In Chapter 3, we used an efficient reduction to flatten a nest to a list so we shall try to use an efficient reduction $\{\![e, f \mid g \mid h]\!\}$ to embed a nest as a list of trees. The necessary types for the parameters are, for some b.

$$\begin{array}{rcl} [e,f] & : & Base\ (b, List\ (Tree\ a)) \to List\ (Tree\ a) \\ g & : & Pair\ b \to b \\ h & : & a \to b \end{array}$$

If these parameters are injective functions, then this efficient reduction is also an injective function, as an embedding function should be. As constructor functions are injective, we choose $b = Tree\ a$ and embed a nest with

$$\begin{array}{rcl} embednest & : & Nest \dot{\to} List \cdot Tree \\ embednest & = & \{\![Nil_{Tree}, Cons_{Tree} \mid Bin \mid Tip]\!\}_{NestF} \end{array}$$

In Haskell, we write (using built-in lists for convenience)

$$\begin{array}{rcl} embednest & :: & Nest\ a \to [Tree\ a] \\ embednest & = & efoldnest\,[\,]\ (uncurry\,(:))\ Tip\ Bin \end{array}$$

8.4 Embedding bushes

Similarly, the embedding function for bushes is given by

$$\begin{array}{rcl} embedbush & : & Bush \dot{\to} U \cdot V \\ embedbush & = & \{\![e, f \mid g_1 \mid g_2 \mid h]\!\}_{BushF} \end{array}$$

The embedding target $U \cdot V$ is yet to be determined. So too are the parameters, which are injective functions with types

$$\begin{array}{rcl} [e,f] & : & Base\ (V\ a, U\ (V\ a)) \to U\ (V\ a) \\ g_1 & : & Outr\ (V\ a, U\ (V\ a)) \to V\ a \\ g_2 & : & V\ a \to V\ a \\ h & : & a \to V\ a \end{array}$$

We can now choose U and V to have constructor functions that can be parameters of the efficient reduction. As before $[e, f]$ can be $[Nil_V, Cons_V]$

so $U = List$. Now V can be chosen so that its constructor functions are g_1 and g_2 and h. Then the embedding target of *Bush* is $List \cdot Thing$ where *Thing* is defined by

$$Thing\ A \quad \approx \quad A + List\ (Thing\ A) + Thing\ A$$

Note that the recursion in the definition of *Bush* is such that we could not have used the technique demonstrated in the last section to derive the embedding target. In Haskell, showing the constructor functions, we write

$$\begin{array}{lcl} \textbf{type}\ BushR\ a & = & [Thing\ a] \\ \textbf{data}\ Thing\ a & = & One\ a \mid More\ [Thing\ a] \mid Extra\ (Thing\ a) \end{array}$$

The embedding function is therefore

$$\begin{array}{lcl} embedbush & : & Bush \dot{\to} List \cdot Thing \\ embedbush & = & \{[Nil_{Thing}, Cons_{Thing} \mid More \mid Extra \mid One]\}_{BushF} \end{array}$$

The constructor function *Extra* is redundant, however, as id_{Thing} is an injective function of the same type, so if we replaced *Extra* with id_{Thing} then *embedbush* would still be an injective function. In Haskell, we can therefore write

$$\textbf{data}\ Thingless\ a \quad = \quad One\ a \mid More\ [Thingless\ a]$$

$$\begin{array}{lcl} embedbush & :: & Bush\ a \to [Thingless\ a] \\ embedbush & = & eredbush\ [\,]\ (uncurry\ (:))\ One\ More \end{array}$$

Here, *eredbush* is *efoldbush* specialised to constant functors. For example, we have

$$\begin{array}{ll} & embedbush\ (ConsB\ (1, ConsB\ (ConsB\ (2, NilB), NilB))) \\ = & [One\ 1, More\ (One\ 2)] \end{array}$$

8.5 Embedding arbitrary datatypes

The generalisation of *embednest* and *embedbush* to an arbitrary datatype T is easy to guess but difficult to write concisely.

$$\begin{array}{lcl} embed_T & : & T \dot{\to} U \cdot V \\ embed_T & = & \{[(\alpha_{UF})_V \parallel \gamma \parallel h]\}_F \end{array}$$

Here, h and the γ_i give the constructors of V, so V is defined according to the types of these parameters. The functors T and U are the least fixed points of hofunctors F and UF respectively, where $F\,X \;=\; B\cdot\langle Id, X\cdot F_1\,X\rangle$. Naturally, this definition of $embed_T$ is too informal to be used in generic proofs. We would like to define a generic operator, to be called the *embedding operator*, that takes the initial algebras of U and V and returns the correct efficient reduction for $embed_T$. For nests, we would have

$$\{\![f \mid h, g]\!\}_{NestF} \;=\; \{\![f \mid g \mid h]\!\}_{NestF}$$

Here the notation "h, g" is shorthand for the join $[h, g]$, which is α_{TreeF} if $h = Tip$ and $g = Bin$. For bushes we have

$$\{\![f \mid h, [g_1, g_2]]\!\}_{BushF} \;=\; \{\![f \mid g_1 \mid g_2 \mid h]\!\}_{BushF}$$

In general we have

$$\{\![f \parallel h, \theta_1\,\gamma]\!\}_F \;=\; \{\![f \parallel \gamma \parallel h]\!\}_F$$

Suppose F_i is defined by

$$F_i\,X \;=\; B_i\cdot\langle Id, X\cdot F_j\,X\rangle$$

Then θ is defined by structural induction as follows.

$$\theta_i\,\gamma \;=\; [\gamma_i, \theta_j\,\gamma]$$

The typings are, for some a,

$$\begin{aligned} \gamma_i &: B_i\;(V\;a, U\;(V\;a)) \to V\;a \\ \theta_i\,\gamma &: \delta_i\;(V\;a,\;U\;(V\;a)) \to V\;a \end{aligned}$$

Here δ is also defined by structural induction.

$$\delta_i\;(Y, Z) \;=\; B_i\;(Y, Z) + \delta_j\;(Y, Z)$$

If B_i is unary, then $\delta_i\;Y = B_i\;Y$ and $\theta_i\,\gamma = \gamma_i$. With a type signature, the generic operator we have just defined is

$$\begin{aligned} \{\![-\parallel-]\!\}_F &: (B\;(b, c) \to c) \to (C_F\;(a, b, c) \to b) \to T\;a \to c \\ \{\![f \parallel h, \theta_1\,\gamma]\!\}_F &= (\![f \parallel \gamma]\!)_F \cdot T\;h \end{aligned}$$

Here, the relator C_F is defined by $C_F\ (X, Y, Z) = X + \delta_1\ (Y, Z)$. Now the embedding function is given simply by

$$\begin{array}{lcl} \mathit{embed}_T & : & T \dot{\to} U \cdot V \\ \mathit{embed}_T & = & \{\![(\alpha_{UF})_V \mid \alpha_{VF}]\!\}_F \end{array}$$

Note that the parameters to the embedding operator need not be initial algebras; the embedding function is a special case where $b = V\ a$. The embedding target of T is $U \cdot V$ where U and V are least fixed points of UF and VF defined by

$$\begin{array}{rcl} UF\ X & = & B \cdot \langle Id, X \rangle \\ VF\ X & = & B' \cdot \langle Id, X \rangle \\ B'\ (X, Y) & = & C_F\ (X, Y, U\ Y) \end{array}$$

Chris Okasaki also proposed similar laws for calculating the embedding target of a nested datatype, but they were neither motivated nor verified [Oka98a].

8.6 Fold-fusion law for embedding operator

In the next section, we shall require a fold-fusion law for the embedding operator we have just derived. Let $R\,(i)$ abbreviate

$$k' \cdot \gamma_i \ = \ \gamma_i' \cdot B_i\ (k', k)$$

We start from the fold-fusion law for efficient reductions that we derived at the end of Chapter 5.

Recall that if we have

$$k \cdot f \ = \ f' \cdot B\ (k', k) \quad \text{and} \quad k' \cdot h \ = \ h'$$

and for all i we have $R\,(i)$ then we conclude

$$k \cdot \{\![f \parallel \gamma \parallel h]\!\}_F \ = \ \{\![f' \parallel \gamma' \parallel h']\!\}_F$$

For $NestF$, the final conjunct is simply

$$k' \cdot \gamma_1 \ = \ \gamma_1' \cdot Pair\, k'$$

For $BushF$, the final conjunct is a pair of conditions that can be written as one.

$$k' \cdot [\gamma_1, \gamma_2] \;\;=\;\; [\gamma_1', \gamma_2'] \cdot (\mathit{Outr}\ (k', k) + \mathit{Id}\, k')$$

In general, we have $P\,(1)$ where $P\,(i)$ abbreviates

$$k' \cdot \theta_i\, \gamma \;=\; \theta_i\, \gamma' \cdot \delta_i\ (k', k)$$

We can replace the clause $R\,(i)$ for all i in the fusion law above with the simpler $P\,(1)$ if we can show it be a sufficient condition. This will give us a very simple fold-fusion law for the embedding operator. The proof is a tedious exercise in structural induction and can safely be skipped by the reader.

Let $Q\,(i)$ mean that $R\,(i)$ holds and $R\,(i')$ holds for all hofunctors $F_{i'}$ that depend on F_i, directly or indirectly. All we need to show is that $P(i)$ implies $Q(i)$ for all i. Suppose that $P\ (i)$ holds and that F_i is defined by $F_i\, X \;=\; B_i \cdot \langle \mathit{Id}, X \cdot F_j\, X\rangle$. Suppose, for an induction hypothesis, that $P(j)$ implies $Q(j)$. If $P(i)$ holds then from the definitions of θ and δ we have

$$k' \cdot [\gamma_i, \theta_j\, \gamma] \;\;=\;\; [\gamma_i', \theta_j\, \gamma'] \cdot (B_i\ (k', k) + \delta_j\ (k', k))$$

The laws of coproducts give us $R(i)$ and $P(j)$, from which we get $Q(j)$ by the induction hypothesis. Then $R(i)$ and $Q(j)$ together give $Q(i)$.

Finally, to neaten the law, we combine $P(1)$ with $k' \cdot h = h'$ to give

$$k' \cdot [h, \theta_1\, \gamma] \;\;=\;\; [h', \theta_1\, \gamma'] \cdot (id + \delta_1\ (k', k))$$

Now we have proved the following law.

Simplified generic fold-fusion law for embedding operator

$$k \cdot \{\![f \parallel g]\!\}_F = \{\![f' \parallel g']\!\}_F$$

$$\Leftarrow\;\; \exists k' : \quad k \cdot f = f' \cdot B\ (k', k) \quad \text{and} \quad k' \cdot g = g' \cdot C_F\ (id, k', k)$$

8.7 Fold-equivalence law

Let *ef* abbreviate the function *efoldnest nil cons tip bin*. Consider the action of *ef* on nests. It replaces the constructor function *NilN* with *nil*. However,

this has the same effect as replacing *NilN* with the empty list [] and then replacing that with *nil*. Applying this idea to the other three constructor functions as well suggests

$$ef \;=\; foldr\ cons\ nil \cdot map\ (foldtree\ tip\ fork) \cdot embednest$$

Here, *foldtree* is the standard fold for the datatype *Tree*. If T and U and V are defined as above then we can now suggest the following generic law, which we shall call the *fold-equivalence law*.

$$(\!| f \mid id |\!)_{UF} \cdot U\ (\!| g' \mid id |\!)_{VF} \cdot embed_T \;=\; \{\![f \mid g]\!\}_F$$

The special case of nests above suggests that $g = g'$ but in fact, g' is a function of g to be derived with the conditions. First we introduce some abbreviations

$$\begin{aligned} k' &= (\!| g' \mid id |\!)_{VF} \\ k &= (\!| f |\!)_{UF} \cdot U\,k' = (\!| f \cdot B\ (k', id) \mid id |\!)_{UF} \end{aligned}$$

Now we prove the law.

$$\begin{aligned}
& k \cdot \{\![\alpha_U \mid \alpha_V]\!\}_F = \{\![f \mid g]\!\}_F \\
\Leftarrow \quad & \{ \text{ fold-fusion law for the embedding operator; } B \text{ is a functor } \} \\
& k \cdot \alpha_U = f \cdot B\ (k', id) \cdot B\ (id, k) \\
& k' \cdot \alpha_V = g \cdot C\ (id, k', k) \\
\equiv \quad & \{ \text{ definition of } k\text{; universal property of standard folds } \} \\
& k' \cdot \alpha_V = g \cdot C\ (id, k', (\!| f \cdot B\ (k', id) \mid id |\!)_{UF}) \\
\equiv \quad & \{ \ C \text{ is a functor; map-fusion law of standard folds } \} \\
& k' \cdot \alpha_V = (g \cdot C\ (id, id, (\!| f \mid id |\!)_{UF})) \cdot C\ (id, k', U\,k') \\
\equiv \quad & \{ \text{ definition of } C \ \} \\
& k' \cdot \alpha_V = (g \cdot C\ (id, id, (\!| f \mid id |\!)_{UF})) \cdot B'\ (id, k') \\
\equiv \quad & \{ \text{ universal property of standard folds } \} \\
& k' = (\!| g \cdot C\ (id, id, (\!| f \mid id |\!)_{UF}) \mid id |\!)_{VF}
\end{aligned}$$

So for the function k' we have just derived, we have

$$(\!| f |\!)_{UF} \cdot U\,k' \cdot \{\![\alpha_U \mid \alpha_V]\!\}_F \;=\; \{\![f \mid g]\!\}_F$$

This proof suggests the following intuition for the fold-fusion law of the embedding operator. To push a function k through an efficient fold on T, we need conditions for pushing k through a standard fold on U-structures of V-structures. This involves pushing some other function k' down a standard fold on a V-structure. When T is non-linear, this itself involves pushing k through a further standard fold on U-structures.

8.7.1 Embedding functions preserve flattening

Now we shall illustrate the fold-equivalence law for the datatype of nests. In Chapter 3, we discovered how to flatten a nest with an efficient reduction.

$$\mathit{eflatnest} \quad = \quad \{\![\mathit{knil}, (:) \mid \mathit{wrap}, (+\!\!+)]\!\}_{\mathit{NestF}}$$

The fold-equivalence law gives

$$((\![\mathit{knil}, (:) \mid \mathit{id}]\!)_{\mathit{ListF}} \cdot \mathit{List}\ (\![\mathit{wrap}, (+\!\!+) \mid \mathit{id}]\!)_{\mathit{TreeF}}) \cdot \mathit{embednest} \quad = \quad \mathit{eflatnest}$$

Each of the folds on trees flattens a tree to a list. The fold on lists flattens the resulting list of lists to a list. The term in brackets consequently flattens a list of trees, so the whole equation asserts that *embednest* preserves flattening. We know that any injective function can be an embedding for nests, but here we have a property that is satisfied by *embednest* but not by *zembednest*. However, the property of preserving flattening is not strong enough to characterise the embedding function. The extra property of *embednest* that we cannot formalise is the way it preserves how exactly the elements are grouped.

8.7.2 Example: summation

Chapter 4 defines the generic summation operator, *sum* but it is motivated by type considerations. This approach is particularly unsatisfactory for non-linear datatypes T because it is difficult to visualise a typical evaluation. A better idea would be to write $\mathit{sum}_{\overline{T}}$ and then use our new law to derive sum_T. For the example of nests,

$$\mathit{sum}_{\mathit{NestR}} \cdot \mathit{embed}_{\mathit{Nest}} \quad = \quad \mathit{sum}_{\mathit{Nest}}$$

Recall how *sum* is defined on compositions:

$$\mathit{sum}_{F \cdot G} \quad = \quad \mathit{sum}_F \cdot F\ \mathit{sum}_G$$

Given this and the fold-equivalence law, we suggest

$$sum_{Nest} \quad = \quad \{\![sum_{Base} \mid sum_{Base'}]\!\}_{NestF}$$

However, $sum_{Base'} = [id, sum_{Pair}]$ so we conclude as before,

$$sum_{Nest} \quad = \quad (\![sum_{Base} \mid sum_{Pair}]\!)_{NestF}$$

8.7.3 Example: unfanning

Chapter 6 defines the polyfunctorial unfanning operation *unfan*. Its composition case is the same as that of *sum*.

$$unfan_{F \cdot G} \quad = \quad unfan_F \cdot F\ unfan_G$$

Once again we motivated the nested fixpoint case of this operation by type considerations, but we could have derived it instead. For nests, we would have

$$unfan_{Nest} \quad = \quad \{\![unfan_{Base} \mid unfan_{Base'}]\!\}_{NestF}$$

To write the second parameter as a join, we first work out $fan_{Base'}$.

$$fan_{Outl + Pair \cdot Outr} \quad = \quad (fan_{Outl} + Pair\ fan_{Outr} \cdot fan_{Pair}) \cdot [id, id]^{\circ}$$

Taking the converse of both sides,

$$fan_{Outl + Pair \cdot Outr}{}^{\circ} \quad = \quad [id, id] \cdot (fan_{Outl}{}^{\circ} + fan_{Pair}{}^{\circ} \cdot Pair\ fan_{Outr}{}^{\circ})$$

Clearly now, $unfan_{Base'}$ is the join $[id, unfan_{Pair}]$ so $unfan_{Nest}$ is a reduction.

$$unfan_{Nest} \quad = \quad (\![unfan_{Base} \mid unfan_{Pair}]\!)_{NestF}$$

Both $unfan_{Nest}$ and sum_{Nest} are Meertens reductions [Mee96], that is, reductions of $T\ a \rightarrow a$, though he considers only regular T. Now we consider another Meertens reduction for the different datatype of power trees.

8.7.4 Example: last

Our final illustration of the fold-equivalence law uses the function *lastpow*, which takes a power tree and returns its rightmost element. There is no

equivalent function for nests because some nests are empty. Recall that the type of power trees was defined in Chapter 1 by

$$\textbf{data}\ Pow\ a\ =\ Zero\ a \mid Succ\ (Pow\ (Pair\ a))$$

Now, *Pow* is what we call a *tail-recursive datatype*, which means that the recursive uses are wrapped by unary constructors. In the notation of category theory,

$$\begin{aligned} Pow &\approx PowF\ Pow \\ PowF\ X &= + \cdot \langle Id, X \cdot Pair \rangle \end{aligned}$$

The embedding target of *Pow* is $Wrap \cdot Tree$ where

$$\begin{aligned} Wrap &\approx WrapF\ Wrap \\ WrapF\ X &= + \cdot \langle Id, X \rangle \end{aligned}$$

The embedding function *embedpow* exchanges the constructors of *Pow* that encode the height of the power tree for constructors of *Wrap* that serve the same purpose. However, the latter constructors are redundant because the tree they wrap has the same type whatever its height.

According to the fold-equivalence law, the last element of a power tree is given by

$$lastpow = \{\![last_{Wrap} \mid last_{Tree}]\!\}_{PowF}$$

Filling in the regular cases is easy.

$$lastpow = \{\![id, id \mid id, outr]\!\}_{PowF}$$

The fold-equality law allows us to rewrite *lastpow* as a simple fold.

$$lastpow = (\![[id, id] \cdot (id + k)]\!)_{PowF}$$

We require that for some k'

$$\begin{aligned} k \cdot [id, id] &= [id, id] \cdot (k' + k) \\ k' \cdot outr &= outr \cdot Pair\ k' \\ k' \cdot id &= outr \cdot Pair\ id \end{aligned}$$

These conditions are solved by $k = k' = outr$ so we have

$$lastpow \;=\; (\![id, outr]\!)_{PowF}$$

However, as the *Wrap* datatype is redundant, a more sensible embedding target for *Pow* might be *Tree*. The embedding function would then be

$$embedpow' \;=\; \{\![id, id \mid \alpha]\!\}_{PowF}$$

Since $[id, id]$ is an injective function, $embedpow'$ is also an injective function. The required variant of the fold-equivalence law is therefore

$$(\![g \mid id]\!)_{TreeF} \cdot embedpow' \;=\; \{\![id, id \mid g]\!\}_{PowF}$$

To prove this we apply the fold-fusion law and get two conditions that are true if $k' {=} (\![g \mid id]\!)_{TreeF}$:

$$\begin{aligned} (\![g \mid id]\!)_{TreeF} \cdot [id, id] &= [id, id] \cdot (k' + (\![g \mid id]\!)_{TreeF}) \\ k' \cdot \alpha &= g \cdot (id + Pair\; k') \end{aligned}$$

Any tail-recursive datatype can be transformed, using an accumulating parameter if necessary, so as to make the constructor functions unary for its base cases.

8.8 Another law for efficient reductions

Unfortunately, the fold-equivalence law and the variant we derived for power trees and other tail-recursive datatypes can only be used with efficient reductions that return regular datatypes. To prove another variant that does not have this restriction, we shall derive a condition connecting g and g' in the following:

$$embedpow' \cdot \{\![id, id \mid g]\!\}_{PowF} \;=\; (\![g' \mid id]\!)_{TreeF} \cdot embedpow'$$

We argue as follows:

$$\begin{aligned} & embedpow' \cdot \{\![id, id \mid g]\!\}_{PowF} = (\![g' \mid id]\!)_{TreeF} \cdot embedpow' \\ \equiv \quad & \{ \text{ first variant of fold-equivalence law } \} \\ & embedpow' \cdot \{\![id, id \mid g]\!\}_{PowF} = \{\![id, id \mid g']\!\}_{PowF} \\ \equiv \quad & \{ \text{ fold-fusion } \} \\ & embedpow' \cdot [id, id] = [id, id] \cdot (k' + \{\![id, id \mid \alpha]\!\}_{PowF}) \end{aligned}$$

$$k' \cdot g = g' \cdot Base'\ (id, k')$$

$$\equiv \quad \{ \text{ universal property of coproducts } \}$$

$$embedpow' \cdot g = g' \cdot Base'\ (id, embedpow')$$

The law we have just derived is

$$embedpow' \cdot \{\![id, id \mid g]\!\}_{PowF} = (\![g' \mid id]\!)_{TreeF} \cdot embedpow'$$
$$\Leftarrow embedpow' \cdot g = g' \cdot Base'\ (id, embedpow')$$

If we write g and g' as the joins $[tip, bin]$ and $[tip', bin']$ then

$$\begin{aligned} embedpow' \cdot tip &= tip' \\ embedpow' \cdot bin &= bin' \cdot Pair\ embedpow' \end{aligned}$$

If we let $[tip', bin']$ be $[Tip, Bin]$ then $[tip, bin]$ are functions on power trees that correspond to functions $[Tip, Bin]$ on trees; the function *bin* was given in Chapter 1 but we shall now derive it after noting that $tip = Zero$. The base case is easy.

$$bin\ (Zero\ x, Zero\ y) \quad = \quad Succ\ (Zero\ (x, y))$$

To derive the inductive case, we use the following laws, which are obtained by using the fold-equality law to rewrite $embedpow'$ as a simple fold; we show how to do this in the next section.

$$\begin{aligned} e \cdot Succ &= Tree\ h \cdot e \\ Succ \cdot e^\circ &= e^\circ \cdot Tree\ h \end{aligned}$$

Here, e abbreviates $embedpow'$ and h abbreviates $[Bin \cdot Pair\ Tip, Bin]$. The derivation of the inductive case of *bin* is as follows:

$$\begin{aligned} & bin \cdot Pair\ Succ \\ = \ & e^\circ \cdot Bin \cdot Pair\ (e \cdot Succ) \\ = \ & e^\circ \cdot Bin \cdot Pair\ (Tree\ h \cdot e) \\ = \ & e^\circ \cdot Tree\ h \cdot Bin \cdot Pair\ e \\ = \ & Succ \cdot e^\circ \cdot Bin \cdot Pair\ e \\ = \ & Succ \cdot bin \end{aligned}$$

In pointwise form we write

$$bin\ (Succ\ x, Succ\ y) \quad = \quad Succ\ (bin\ (x, y))$$

This definition of *bin* appears in [BGJ00] but it is not derived there. However, many functions can be built from *bin* so it is useful to have derived it formally. For example, if *swap* is the natural transformation that swaps a pair, then the function that reverses a tree is

$$\begin{array}{lcl} \mathit{revtree} & : & \mathit{Tree} \dot{\to} \mathit{Tree} \\ \mathit{revtree} & = & (\!|\mathit{id}, \mathit{Bin} \cdot \mathit{swap}_{\mathit{Tree}} \mid \mathit{id}|\!)_{\mathit{TreeF}} \end{array}$$

According to our new variant of the fold-equivalence law, the function that reverses a power tree is

$$\begin{array}{lcl} \mathit{revpow} & : & \mathit{Pow} \dot{\to} \mathit{Pow} \\ \mathit{revpow} & = & \{\![\mathit{id}, \mathit{id} \mid \mathit{id}, \mathit{bin} \cdot \mathit{swap}_{\mathit{Pow}}]\!\}_{\mathit{PowF}} \end{array}$$

Unfortunately, *bin* is partial so *rev* can be partial too. It is not the case that we can write a fold on trees that leaves the output unbalanced and then be alerted to this error when we rewrite the fold for power trees using the law. Although *bin* seems useful for writing programs, none of the programs thus constructed are useful because they are all partial.

8.9 Laws for arbitrary combinators

Recall that we want to work out f from f' or f' from f in the equation below.

$$\mathit{embed}_Y \cdot f = f' \cdot \mathit{embed}_X$$

We have already handled the case where f is an efficient reduction. Now we consider other possibilities for f. If f is an identity function then so is f'. If f is a composition $g \cdot h$ then we need a condition pushing embed_Y through g and a similar condition for h. If f is a map operation then so is f' because embedding functions are proper natural transformations. Embedding functions can be shunted, so we have the following law for dealing with converses.

$$\mathit{embed}_X \cdot f^{\circ} = f'^{\circ} \cdot \mathit{embed}_Y \quad \Leftarrow \quad \mathit{embed}_Y \cdot f = f' \cdot \mathit{embed}_X$$

Pushing *embed* through zips is an application of the higher-order naturality property of zips. For linear T and functional zips we have

$$(\mathit{embed}_T)_G \cdot \mathit{zip}_{G,T} \quad = \quad \mathit{zip}_{G,\overline{T}} \cdot G\ \mathit{embed}_T$$

Suppose we extend the notion of embedding targets with $\overline{T \cdot G} = \overline{T} \cdot G$ and $\overline{G \cdot T} = G \cdot \overline{T}$, for polynomial or regular G. If we also extend the embedding function accordingly then a law for pushing zips through embedding functions.

$$embed_{Nest \cdot G} \cdot zip_{G,Nest} \quad = \quad zip_{G,NestR} \cdot embed_{G \cdot Nest}$$

This property also holds for bushes and we can use it to derive a zip for bushes, but unfortunately such a zip must have a $\overline{Bush}$-structure as an intermediate.

Finally, and most importantly, a law for pushing embedding functions through initial algebras will enable us to deal with constructors in programs. We shall use the fold-equality law to rewrite *embednest* as a simple fold $(\![h]\!)_{NestF}$ where

$$h \quad = \quad (\![\alpha_{List} \cdot Base\ (Tip, List\ (\![Bin \cdot Pair\ Tip, Bin \mid id]\!)_{TreeF}]\!)_{NestF}$$

For some k, we have

$$\begin{array}{lcl} embednest' & : & Nest \dot{\rightarrow} List \cdot Tree \\ embednest' & = & (\![\,[Nil, Cons] \cdot Base\ (Tip, k)]\!)_{NestF} \end{array}$$

The conditions on k are that, for some k',

$$\begin{array}{rcl} k \cdot [Nil, Cons] & = & [Nil, Cons] \cdot Base\ (k', k) \\ k' \cdot Bin & = & Bin \cdot Pair\ k' \\ k' \cdot Tip & = & Bin \cdot Pair\ Tip \end{array}$$

If we combine the last two equations then the universal property of standard folds gives us

$$k' \quad = \quad (\![Bin \cdot Pair\ Tip, Bin \mid id]\!)_{TreeF}$$

Using the universal property again we get

$$k \quad = \quad (\![\,[Nil, Cons] \cdot Base\ (k', id) \mid id]\!)_{ListF}$$

However, this is equal to $List\ k'$ so we get the following expression for $embednest'$.

$$\begin{array}{rcl} embednest' & : & Nest \dot{\rightarrow} List \cdot Tree \\ embednest' & = & (\![h]\!)_{NestF} \\ h & = & [Nil, Cons] \cdot Base\ (Tip, List\ (\![Bin \cdot Pair\ Tip, Bin \mid id]\!)_{TreeF}) \end{array}$$

Then a law for initial algebras is given by the definition of simple folds.

$$(\!|h|\!)_{NestF} \cdot \alpha_{NestF} \quad = \quad h \cdot Base\ (id, ((\!|h|\!)_{NestF})_{Pair})$$

We can generalise this law to linear datatypes but not to non-linear datatypes because we need to use the fold-equality law to derive it. Now we have the ability to remove nests from programs. However, this power is not as useful as we might imagine because we cannot always remove polymorphic recursion. We shall demonstrate this by deriving the function *hfoldnestr* on lists of trees that corresponds to *hfoldnest* on nests. We start by deriving two conditions that are equivalent to and that can be used instead of the following:

$$embed_Y \cdot f \quad = \quad f' \cdot embed_X$$

Since the embedding function is injective and not surjective, we have

$$\begin{array}{lcl} embed_T{}^\circ \cdot embed_T & = & id_T \\ embed_T \cdot embed_T{}^\circ & \subseteq & id_{\overline{T}} \end{array}$$

Now we can apply $embed_Y{}^\circ$ to both sides and get

$$f \quad = \quad embed_Y{}^\circ \cdot f' \cdot embed_X$$

Alternatively, we can apply $embed_X{}^\circ$ to both sides and get

$$embed_Y \cdot f \cdot embed_X{}^\circ \quad \subseteq \quad f'$$

This can be strengthened to an equality if we are prepared to restrict the domain of *hfoldnestr* to lists of trees that obey the constraint of being "nest-like".

$$\begin{array}{ll}
 & hfoldnestr\ f' \\
= & \left\{ \begin{array}{l} \text{fact about embedding functions;} \\ \text{link between } f \text{ and } f' \text{ derived later} \end{array} \right\} \\
 & embed_Y \cdot (\!|f|\!)_{NestF} \cdot embed_{Nest}{}^\circ \\
= & \left\{ \text{definition of simple fold operator with } \alpha_{NestF} \text{ shunted} \right\} \\
 & embed_Y \cdot f \cdot Base\ (id, ((\!|f|\!)_{NestF})_{Pair}) \cdot \alpha^\circ \cdot embed_{Nest}{}^\circ \\
= & \left\{ \begin{array}{l} \text{see above; axioms of converse;} \\ Base \text{ is a functor; definition of simple fold operator} \\ \textbf{assume: } embed_Y \cdot f = f' \cdot Base\ (id, embed_{Y \cdot Pair}) \end{array} \right\} \\
 & f' \cdot Base\ (id, (embed_Y)_{Pair} \cdot ((\!|f|\!)_{NestF})_{Pair}) \cdot ((embed_{Nest})_{Pair})^\circ) \cdot h^\circ
\end{array}$$

$$
\begin{array}{ll}
= & \left\{ \begin{array}{l} \text{same fact about embedding functions; definition of } h \\ \textbf{let } \mathit{thing} = (\![\mathit{Bin} \cdot \mathit{Pair\ Tip}, \mathit{Bin} \mid \mathit{id}]\!)_{\mathit{TreeF}}{}^{\circ} \end{array} \right\} \\
& f' \cdot \mathit{Base}\ (\mathit{Tip}^{\circ}, (\mathit{hfoldnestr}\, f')_{\mathit{Pair}} \cdot \mathit{List\ thing}) \cdot \alpha_{\mathit{ListF}}{}^{\circ}
\end{array}
$$

In summary, we have

$$
\begin{array}{rcl}
\mathit{hfoldnestr}' & : & (\mathit{NestF}\ R \dot{\to} R) \to (\mathit{NestR} \dot{\to} R) \\
\mathit{hfoldnestr}\, f' \cdot \alpha_{\mathit{ListF}} & = & f' \cdot \mathit{Base}\ (\mathit{Tip}^{\circ}, (\mathit{hfoldnestr}\, f')_{\mathit{Pair}} \cdot \mathit{List\ thing}) \\
\mathit{thing} & = & (\![\mathit{Bin} \cdot \mathit{Pair\ Tip}, \mathit{Bin} \mid \mathit{id}]\!)_{\mathit{TreeF}}{}^{\circ}
\end{array}
$$

8.10 Nested relators have membership

Oege de Moor has pointed out to the author that the existence of an embedding function for a nested functor is all that is needed to show that the functor has membership. We define the candidate membership as

$$
\in_T \quad = \quad \in_{\overline{T}} \cdot \mathit{embed}_T
$$

If this really is a membership then it must be equal to the membership of Chapter 6 because memberships are unique but it is difficult to show this in any other way because the membership of Chapter 6 is not expressed as a fold. To show that $\in_T$ satisfies the characterisation of membership we begin by stating that $\in_{\overline{T}}$ does.

$$
\begin{array}{ll}
& \overline{T}\, R \cdot (\in_{\overline{T}} \backslash \mathit{id}) = (\in_{\overline{T}}) \backslash R \\
\equiv & \left\{ \ \mathit{embed}_T{}^{\circ} \text{ is a partial function } \right\} \\
& \mathit{embed}_T{}^{\circ} \cdot \overline{T}\, R \cdot (\in_{\overline{T}} \backslash \mathit{id}) = \mathit{embed}_T{}^{\circ} \cdot (\in_{\overline{T}} \backslash R) \\
\equiv & \left\{ \ \mathit{embed}_T{}^{\circ} \text{ is a proper natural transformation } \right\} \\
& T\, R \cdot \mathit{embed}_T{}^{\circ} \cdot (\in_{\overline{T}} \backslash \mathit{id}) = \mathit{embed}_T{}^{\circ} \cdot (\in_{\overline{T}} \backslash R) \\
\equiv & \left\{ \ \text{law of division on both sides; } \mathit{embed}_T \text{ is a function } \right\} \\
& T\, R \cdot ((\in_{\overline{T}} \cdot \mathit{embed}_T) \backslash \mathit{id}) = ((\in_{\overline{T}} \cdot \mathit{embed}_T) \backslash R) \\
\equiv & \left\{ \ \text{definition of } \in_T \ \right\} \\
& T\, R \cdot (\in_T \backslash \mathit{id}) = \in_T \backslash R
\end{array}
$$

8.11 Embedding predicates

An embedding predicate takes a member of the embedding target and tests whether it is a member of the target of the embedding function. Embedding predicates are extensively studied in Borges' thesis [Bor01]. We will derive the embedding predicate for the datatype of non-empty nests defined below. It may be possible, by finding out what properties of $Base^+$ are actually used, to generalise the derivation below to other linear datatypes, but unfortunately it does not appear that our favourite datatype of nests is one of them.

$$\begin{aligned} Nest^+ &\approx NestF^+\, Nest^+ \\ NestF^+\, X &= Base^+ \cdot \langle Id, X \cdot Pair \rangle \\ Base^+\ (X, Y) &= X + X \times Y \end{aligned}$$

The embedding target of $Nest^+$ is $NestR^+ = List^+ \cdot Tree$ where $Tree$ is defined as before and $List^+$ is defined by

$$\begin{aligned} List^+ &\approx ListF^+\, List^+ \\ ListF^+\, X &= Base^+ \cdot \langle Id, X \rangle \end{aligned}$$

Formally, we shall derive a function *perfect* that satisfies the equation.

$$perfect \cdot embed_{Nest^+} = ktrue$$

Here, *ktrue* is the constant function that always returns the boolean value true. We expect that the function *perfect* will traverse a list of trees, comparing heights of adjacent trees, so we shall refine it as follows.

$$perfect = iszero \cdot ok$$

The function $ok : NestR^+ \dot{\to} K_{Int}$ returns the integer 0 when it is given a non-empty list of trees that corresponds to a non-empty nest. The function $iszero : K_{Int} \dot{\to} K_{Bool}$ returns the integer 0 when given the boolean value true and is undefined otherwise. We state the equation below because the left-hand side is equal to $(\![ktrue, outr]\!)_{NestF}$ (by the universal property) and the right-hand side is also equal to $(\![ktrue, outr]\!)_{NestF}$ (by the fold-fusion law).

$$ktrue = iszero \cdot (\![kzero, outr]\!)_{NestF}$$

Using our equations for *perfect* and *ktrue*, we rewrite our requirement to

$$ok \cdot embed_{Nest^+} = (\![kzero, outr]\!)_{NestF}$$

By the fold-equivalence law, ok is given by

$$ok \;=\; (\![e, f \mid id]\!)_{ListF^+} \cdot List^+\,(\![h, g \mid id]\!)_{TreeF}$$

Here, we have assumed that the simple fold $(\![kzero, outr]\!)_{NestF}$ can be rewritten using the fold-equality law to the efficient reduction $\{\![e, f \mid g \mid h]\!\}_{NestF}$. To apply the fold-equality law, we first rewrite the argument of the simple fold. For some k, we have

$$[kzero, outr] \;=\; [e, f] \cdot (h + h \times k)$$

Now the conditions of the fold-equality law are that, for some k',

$$\begin{aligned} k \cdot [e, f] &= [e, f] \cdot (k' + k' \times k) \\ g \cdot (k' \times k') &= k' \cdot g \\ g \cdot (h \times h) &= k' \cdot h \end{aligned}$$

Using the coproduct fusion laws we conclude that $e = id$ and $h = kzero$ and $k' = k$. Now we only have to solve the following conditions

$$\begin{aligned} outr &= f \cdot (kzero \times k) \\ k \cdot f &= f \cdot (k \times k) \\ g \cdot (k \times k) &= k \cdot g \\ g \cdot (kzero \times kzero) &= k \cdot kzero \end{aligned}$$

Expressed pointwise these conditions are

$$\begin{aligned} f\,(0, k\,y) &= y \\ f\,(k\,x, k\,y) &= k\,(f\,(x, y)) \\ g\,(k\,x, k\,y) &= k\,(g\,(x, y)) \\ g\,(0, 0) &= k\,0 \end{aligned}$$

Choosing $k\,x = x + 1$ gives the following definition for *perfect*.

$$perfect \;=\; iszero \cdot (\![id, asc \mid id]\!)_{ListF^+} \cdot List^+\,(\![kzero, bal \mid id]\!)_{TreeF}$$

Here, *asc* and *bal* are undefined except in the cases

$$\begin{aligned} asc\ (x, x+1) &= x \\ bal\ (x, x) &= x + 1 \end{aligned}$$

Clearly, the fold on trees replaces every perfect tree with its height; it is undefined on imperfect trees. The fold on lists test whether the result of doing this on every tree in the list is $[0, 1, 2, \ldots]$. The function *perfect* returns the boolean value true when given a list of perfect trees with heights starting at 0 and increasing by 1 each time. It is undefined when given anything else.

Chapter 9

Conclusion

In this thesis, we defined some important generic operations on nested datatypes and then proved, using the universal properties and fusion laws of folds, that these operations satisfy certain properties. For example we proved:

- the fold-equality law, which says when a simple fold and an efficient reduction are equal;
- that the zip operation for linear nested datatypes satisfies Hoogendijk's requirements for zips;
- that nested relators have membership, so nested datatypes are container types;
- the fold-equivalence law, which uses the embedding function to relate folds on nested datatypes with folds on their embedding targets;
- nested functors on **Fun** extend to nested relators on **Rel**. This result is crucial because it enables us to reason in an endorelator category augmented by the operators of the relational calculus, which are lifted from the allegory **Rel**.

Indeed, the whole thesis can be seen as an investigation of the universal properties and fusion laws of generalised folds and efficient folds, since everything proved in the thesis follows from these theorems. Our first task in this conclusion will be to review the thesis from that perspective. In particular, it is interesting to see how useful these laws are compared with their counterparts for standard folds. That is why we set ourselves goals, like showing nested

relators have membership, that we know are achievable for standard folds.

We shall then recall our framework for generic reasoning with fold operators and explain why it was not unduly complex. After that we shall propose future work, treating Chapter 8 separately with a detailed discussion that also summarises the chapter and its conclusions.

The universal property of simple folds has exactly the same form as the universal property of standard folds; only the category is different. It follows that many theorems proved for standard folds also hold for simple folds. For example, it was trivial to prove a hylomorphism theorem for simple folds and to prove that initial algebras in **Coc** (**Fun**) are also initial algebras in **Cor** (**Rel**), once we had lifted the laws of power allegories. We also used the universal property of simple folds to prove both the fold-equality law and a law connecting initial algebras of nested datatypes with their embedding functions. These applications are arguably more interesting because they are not simply extensions of theorems for standard folds.

The universal property of generalised folds was used to prove two properties of zips and to show, using the Knaster-Tarski theorem, that the generalised fold operator preserves injectivity of relations. Unfortunately, generalised folds do not have a hylomorphism theorem and they cannot be used zip bushes. We derived a definition of the efficient fold operator from the definition of the generalised fold operator, but we were not able to prove that the former had a universal property merely from the fact that the latter did. Instead we had to show this directly.

The universal property of efficient folds defines a unique mapping on arrows rather than a unique arrow, as is the case with other folds. This mapping itself depends on an existentially quantified collection of mappings on arrows. It is not surprising then that the universal property is difficult to grasp; it is also a source of some disappointment for us as we could not apply the Knaster-Tarski theorem to it in order to prove a hylomorphism theorem. Also we could not derive fusion laws from the universal property directly, nor could we use it to prove the fold-equality law.

However, we were able to prove both that nested functors in **Coc** (**Rel**) commute with converse (making them objects of the endorelator category)

and that generalised unfans are efficient folds. We noted that it was a good idea to rehearse the proofs with nests first since some complex pattern matching was required in the general case.

The operators for efficient folds and generalised folds can each be implemented in terms of the other. Given what we have seen in this thesis, which operator should be regarded as the more fundamental? Choosing the efficient fold operator seems best because then we can implement the map operator without using explicit recursion. The fact that efficient folds are sometimes more efficient is not important when making this decision because this advantage only applies to a few datatypes.

It was Hinze [Hin99a] who discovered that a power tree can be flattened either with a simple fold or with an efficient reduction. The fold-equality law is a significant result because it allows us to convert from one type of fold to the other according to our preferences: simple folds are easier to understand operationally but efficient reductions have simpler type signatures.

The map-fusion law for generalised folds is used in three places: once to prove that zips are higher-order natural, once to prove a fold-fusion law for efficient folds, and once to prove an (uninteresting) map-fusion law for efficient folds. Reductions are closed under map-fusion and that is why the fold-equality law, which looks at first sight to be an application of map-fusion, cannot be proven that way. This illustrates how much harder it is to keep track of typings for such laws when working in a category whose arrows are natural transformations.

The map-fusion law for generalised folds has conditions, unlike its counterpart for standard folds. These conditions could not be met for a certain use of map-fusion that we encountered while retracing Hoogendijk's proof that zips are proper natural transformations; recall that Hoogendijk writes zips as standard folds whereas we write zips as generalised folds. We had to exploit the fact that zips are arrows in the endorelator category. In general, we cannot always hope to find such convenient extra properties with which to salvage proofs and the map-fusion law for generalised folds is, in conclusion, disappointingly limited.

The map-fusion law for efficient folds has no conditions. We used it to prove

that our candidate membership for nested relators satisfies the characterisation of membership. The proof closely mimics the already existing proof for standard folds. The map-fusion law appears therefore to be a major selling point of efficient folds over generalised folds. Indeed it is but the reader can easily be misled about its power. It is not the case that one can perform map-fusion on any generalised fold simply by writing it as an efficient fold. The law for generalised folds ought to be (and yet is not) a special case of the law for efficient folds so the latter is evidently not as general as it could be.

We therefore attempted to derive in three different ways a suitable generalisation of the law for generalised folds. Our three approaches were based on, respectively, the corresponding law for generalised folds, the universal property of efficient folds and the fold-fusion law of efficient folds (where the efficient fold to be fused is the map operation). However, no applications have yet been found for any of the resulting laws.

The restricted nature of these fold-fusion laws has a big impact on our proofs that nested functors in **Coc** (**Fun**) extend to nested relators in **Cor** (**Rel**) and that efficient fold operators in **Coc** (**Fun**) extend to efficient fold operators in **Cor** (**Rel**). We would like to use as few properties of **Rel** as possible. That way the proof will extend to as many allegories as possible. As it happens, we do not know any interesting allegories other than **Rel**, for the purposes of program calculation, but we would still prefer for aesthetic reasons to use only the axioms of categories and allegories. Unfortunately, it appears that we must use both the fact that **Rel** is tabular and the fact that **Rel** is a power allegory.

As we have already explained, the fold-fusion law for simple folds has the same form as the fold-fusion law for standard folds. This is a useful fact to bear in mind when attempting proofs. For example, Hoogendijk used fold-fusion to show that the standard fold operator preserves total relations. This suggests (correctly) we can prove in the same way that the simple fold operator preserves total relations as well.

However, we cannot prove that the output of a generalised fold is always total whenever the main parameter is, because the identity function is a simple fold and the type of the fold-fusion law dictates that only simple folds yield a simple fold when fold-fusion law is applied. This is the opposite of

what might be expected: surely a generalised fold, being more general than a simple fold, is more likely to be the result of fold-fusion, not less? Our experience in this thesis is that fold-fusion laws for generalised folds are less useful than their counterparts for standard folds, just as was the case with map-fusion laws, as we shall now explain.

For a fold of type $T \cdot M \dot{\to} R$ and non-linear T, the condition for fold-fusion contains collections of terms $M\, p_i$ and $R\, p_i$. We cannot think of values for the p_i so we let M and R be constant functors and arrive at a concise law for reductions. From this follows an even more concise law for efficient reductions, which is used in Chapter 8. For linear datatypes, we can also get a more concise law by setting the only p_i to be the identity.

Perhaps the most difficult part of the thesis for the reader to grasp is the framework we introduce in Chapter 3 for generic proofs involving generalised folds and efficient folds. Although we motivated each aspect as it was introduced, the use of hindsight gleaned from examples is a better way to satisfy ourselves that our framework was no more complicated than necessary and that it is indeed an important original contribution. This is what we shall do now, covering linear datatypes and non-linear datatypes separately.

When proving properties of zips, Hoogendijk avoids case analysis by writing each regular functor T in the form $T\ A \approx B\ (A, T\ A)$ for some bifunctor B. Similarly, we avoid case analysis when adapting his proofs by writing each linear T in the form $T \approx B \cdot \langle Id, T \cdot Q \rangle$, for some bifunctor B and endofunctor Q. The uses of the universal property of generalised folds are then similar to, and only slightly more complex than, the uses of the universal property of standard folds. To be sure, the form excludes datatypes, such as de Bruijn terms, that can recurse in two different ways but it is easy to adapt proofs to incorporate them: we simply need to replace the binary functor B with a ternary functor. Indeed we can adapt our working to n-ary functors for any n. We mentioned in Chapter 3 that there was a tradeoff between generality and readability but no new insights on this issue arose during the course of the thesis.

The functor *Bush* is defined to be isomorphic to a functor expression constructed from the composition *Bush*$\cdot$*Bush*. Its generalised fold operator must fold each occurrence of *Bush* separately so it has a different form from the

generalised fold operator for *Nest*. Therefore, we shall also write the fixed point isomorphism in a form different from that of *Nest* and other linear functors.

We introduced a single-case grammar following [BP99b], to describe all the possible forms. We use this grammar to define operations by induction on the structure of polynomial hofunctors without using case analysis. The advantage of this is seen in [BP99b] itself when the fusion laws are derived, and the proofs in this thesis reinforce the point.

The number of parameters to the generalised fold operator varies with the datatype. Both [BP99b] and [BGM] need to use ellipses when writing the auxiliary parameters of a generalised fold for an arbitrary nested functor. This notation is unsightly but that would be forgivable if it were not for the greater problem that we cannot dictate the values for each auxiliary parameter when we specify generic operations as generalised folds. Our solution is to index by the same set both the auxiliary parameters and the subsidiary hofunctors. If we do this we can then we can describe succinctly how exactly the value of each parameter depends on its type and hence on which parameter it is. The benefit of this indexing was demonstrated when we proved that generalised unfans are efficient folds.

The generic operation is defined by induction so we use induction to reason about it. That such proofs are easy to write was demonstrated when we derived both the improved fusion laws and the efficient fold operator. It is easy to adapt these proofs for hofunctors that are constructed from n-ary, rather than binary, polynomial functors. It is important to note that these proofs have no case analysis, whereas Hinze's derivation of the efficient fold operator has a five-way case analysis.

A criticism can be made, however, that the motivation of the generalised fold operator is unconvincing and unduly complex. In particular, Hinze [Hin99a] motivates his first generalised fold operator (for power trees) not by type considerations but by explaining its purpose: the extra parameter is a replacement for the constructor that turns types into pairs. Consequently, his transition from linear to non-linear datatypes is silent, whereas ours takes some explanation. Nevertheless, the generalised folds we use are different from those of Hinze and it is unclear whether Hinze's motivation can be ap-

plied to our generalised folds. Also, it may not be possible to define generic operations such as membership and embedding functions in terms of Hinze's folds.

Finally, we separated out the conditions of the generic fusion laws in [BP99b] into separate conjuncts that make no mention of either folds or maps on the datatype. The benefits of this simplification are clear in the generic laws that follow from the new fusion laws. They culminate in the fold-equivalence law of Chapter 7.

Now we point out the shortcomings of this thesis and suggest possible future work.

- We have mentioned higher-order nested datatypes only briefly in this thesis, and that was when we wrote in Haskell a simple fold operator for square matrices. However, we could not supply to the operator, parameters for flattening a square matrix, nor for any other interesting operations. Furthermore, it appears that the Haskell type class system does not permit us to define a generalised fold operator. However, we can at least try reasoning about folds on square matrices categorically and then implementing them by explicit recursion.

  It should be noted that the style of proofs favoured by Hinze in [Hin00d] can be used to show theorems for arbitrary kinds. For example, Hinze defines a generic map operator for arbitrary kinds and proves that it preserves identities and composition. However, Hinze's proofs are based on logical relations and they assume a domain-theoretic structure on which to conduct fixed point induction, whereas we do not, and our proofs are simpler as a result.

  We have managed to prove everything we wished to, and it is hard to see how we could have done this without the use of category theory, because many of our proofs are based on the universal properties of folds. We also found that the fold operators were of great help because of the structured recursion they offered.

- Recapping what we learned in Chapter 5, a hylomorphism is a fold after an unfold. A hylomorphism theorem says that a hylomorphism can be written as a least fixed point. We concluded that a hylomorphism

theorem for simple folds has to exist simply because a hylomorphism theorem exists for standard folds.

However, generalised hylomorphisms are not fixpoints and efficient hylomorphisms are, in a sense, higher-order fixpoints but we do not know how to show that they are least fixed points. Therefore we can conjecture a hylomorphism theorem for efficient folds but we know there cannot be a hylomorphism theorem for generalised folds. It may be possible to prove the former by extending the Knaster-Tarski theorem to efficient folds, if this even is possible.

Applications of hylomorphism theorems include algorithms that build up and then consume some intermediate data structure written as a nested datatype. It is hard to come up with concrete examples of such hylomorphisms though due to our inexperience with unfolds on nested datatypes. The paper [GJ98], though written for regular datatypes, might help us.

Connected with this is the notion of coinductive datatypes. These are used to represent infinite data structures such as infinite lists or, to pick a more relevant example, the infinite trie structures of [Hin00b]. Whereas the inductive nested datatypes in this thesis have a simple fold operator defined as part of their semantics, a coinductive nested datatype has a simple unfold operator that follows immediately from its definition. This operator can perhaps be used to build a trie structure, from a list of the elements it must contain.

- We sketched how the τ-Δ calculus can be used to redo for multirelators the proofs in the chapters on zips and membership. Perhaps we can find a more readable notation than the τ-Δ calculus to use in its place.
- Wadler explains in [Wad89] how to construct free theorems for any polymorphic function. Two special cases of the free theorem for *foldr* are the fold-fusion and map-fusion laws for lists. It would be interesting to see whether fusion laws can be obtained in the same way from the generalised fold or efficient fold operators. They may be simpler than the laws in [BP99b] or if they are identical then at least they would then be motivated more clearly than at present.

It does appear possible to derive a free theorem for reductions but not for the more general case of generalised fold operations, which have rank two type signatures. The reason why this causes a problem can be understood if we look at the rules in [Wad89]. A new variable is introduced in the fusion conditions for each universal quantifier, so there are too many variables for us to be able to solve the conditions.

- We were not able to implement fans because they are non-deterministic. However, we can simulate non-determinism in a functional language by using a random number generator. We would particularly like to implement fans for non-linear datatypes such as *Bush* because it is takes some thought to generate non-trivial examples of bushes by hand as the recursion involved is so bewildering. This would make it easier to test programs that consume bushes. The fan operation can then be given to the utility Quick Check [CH00], which uses such input randomisers to test programs automatically.

- Now that we have considered the fusion laws of [BP99b] in depth, we can examine the fusion laws for Blampied's folds [Bla00]. Blampied discusses fusion laws in the conclusion of his thesis, where he suggests that the laws must be phrased in terms of the way in which argument families are constructed. As we explain in Chapter 2, Blampied constructs fusion laws in an ad hoc way. However, he suggests in his conclusion that the use of combinators for constructing algebra families may give more general fusion laws.

- The scan operation [BdMH96] on trees labels each node with the result of folding the subtree rooted at that node. It can be generalised to any regular datatype by generalising subtrees to substructures. However, it is difficult to generalise the notion of substructures to nested datatypes. Each of the imperfect possibilities gives a different scan operation.

- We should find out whether our embedding predicate can be generalised beyond non-empty nests.

We close the chapter with some ideas for future work that arise from the previous chapter on embedding, but let us summarise that chapter first. Our starting point was the embedding target given for nests given in [Oka98b].

We defined an embedding function for nests, generalised it to an efficient fold that took for its parameters the constructors of the embedding target, and deduced an embedding target for each nested datatype. In doing so we motivated Okasaki's rules [Oka98a] for constructing embedding targets.

Borges suggests in [Bor01] a different embedding target and embedding function. We gave many reasons for preferring our own but a final reason comes from hindsight: the many useful laws that could be proven with our version. We also proved that our embedding function preserves flattened structure. We noted that the output of embedding is not only isomorphic, but has a structure that would match exactly the input if the constructor functions were removed. This may be enough to characterise the embedding but unfortunately, we do not know how to formalise it.

The embedding function is a simulation relation between programs constructed using nested datatypes and programs constructed using regular datatypes. We can use it to calculate for each program, a program free of nested datatypes that simulates it. Our most general result for this is a function that simulates constructor functions. It allows us to remove nested datatypes from programs. Unfortunately, we cannot always remove polymorphic recursion at the same time. This was illustrated when we rewrote for lists of trees the simple fold operator for nests.

The fold-equivalence law states that an efficient reduction on nests is simulated by a fold on lists after a map with a fold on trees, an operation that does not feature polymorphic recursion. This result would be a little more pleasing, however, if the efficient reduction operator we constructed were not quite so contrived.

More interesting results were obtained by concentrating on the special case of tail-recursive datatypes. Our running example was the datatype *Pow* of power trees. It illustrates what is so special about tail-recursive datatypes: the embedding target identified by Okasaki's rules is a composition of two datatypes that is also isomorphic to the single datatype *Tree* of leaf-labelled trees.

We therefore decided to make the embedding target of *Pow* be *Tree* instead and changed the embedding function accordingly. Once we did this,

we could derive a law asserting that efficient reductions on *Pow* are simulated by standard folds on *Tree*. This law is simpler than the fold-equivalence law but it is restricted to functions that have inputs and outputs of equal type. With this law we derived the partial functions *bin* and *tip* that are simulated by the constructors of *Tree*. It may be possible to do this for other datatypes like that of AVL Trees.

Given these partial functions, we can construct an efficient reduction to reverse a power tree from the standard fold that reverses a tree. In other words, we can write a program for a nested datatype by exploiting our greater familiarity with regular datatypes. This could have been the basis of a very useful programming method. Unfortunately, the programs we get are not guaranteed to terminate as they are constructed from partial functions. Worse still, the partial functions are written using explicit recursion rather than fold or map operators so we do not know of any rules for eliminating them.

In Chapter 1, we explained how to define power trees in Dependent ML by constraining the datatype of leaf-labelled trees such that for each node, both immediate subtrees have the same height. It is arguably easier to specify power trees as constraints in Dependent ML than as nested datatypes — the difference is striking for more complicated datatypes [Xi99]. If we could automatically convert from one to the other, then designing nested datatypes would become much easier than it is now.

One approach might be to link Dependent ML refinements to Hinze's recursive bag equations [Hin01] since Hinze has already connected the latter to nested datatypes. Xi has shown how to write other nested datatypes like Braun trees, which have size constraints instead of height constraints, in Dependent ML as well [Xi99]. It seems possible to capture all tail-recursive datatypes but not any other datatypes. For example, it does not seem possible to specify as a refinement of lists of trees, the constraint of being nest-like.

However, we can ask ourselves how much of a restriction this really is. Nests have a tail-recursive variant, given by the type *NestIter* in Chapter 7, so nests can be designed using the process we have mentioned. Hinze shows in [Hin01] that every regular datatype can be written in a tail-recursive form but this may not be true of nested datatypes: the datatype of de Bruijn terms appears to be a counterexample.

Bibliography

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM Press, 1998.

[BB00] Kevin Backhouse and Roland Backhouse. Private communication, 2000.

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[BdMH96] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[BGB] Richard Bird, Jeremy Gibbons, and Ian Bayley. Pattern-matching in perfect trees. Unpublished.

[BGJ00] Richard Bird, Jeremy Gibbons, and Geraint Jones. Program optimisation, naturally. In *Millenial Perspectives in Computer Science*. Palgrave, 2000.

[BGM] Ian Bayley, Jeremy Gibbons, and Clare Martin. Disciplined, efficient, generalised folds for nested datatypes. Forthcoming.

[Bir89] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science*, volume F55, pages 151–216. Springer-Verlag, New York, NY, 1989.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

[BJJM99] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. In *LNCS*, volume 1608, pages 28–115. Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[Bla00] Paul Blampied. *Structured Recursion for Non-Uniform Data-Types*. PhD thesis, University of Nottingham, 2000.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422, pages 52–67. Springer-Verlag, Berlin, 1998.

[Bor01] Pedro Borges. *On nested datatypes and their regular counterparts*. PhD thesis, University of Oxford, 2001.

[BP99a] Richard Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, January 1999.

[BP99b] Richard Bird and Ross Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.

[FP91] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, 1991.

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings 3rd ACM SIGPLAN Int. Conf. on Functional*

Programming, ICFP'98, Baltimore, MD, USA, 26–29 Sept 1998, pages 273–279. ACM Press, New York, 1998.

[HB97] P. Hoogendijk and R. Backhouse. When do datatypes commute? *Lecture Notes in Computer Science*, 1290:242–260, 1997.

[HdM00] Paul F. Hoogendijk and Oege de Moor. Container types categorically. *Journal of Functional Programming*, 10(2):191–225, 2000.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[Hin99a] Ralf Hinze. Efficient generalized folds. In *Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal, 6th July 2000*, 1999.

[Hin99b] 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 *LNCS*, pages 21–36. Springer, November 1999.

[Hin00a] Ralf Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming*, 10(3):305–317, May 2000.

[Hin00b] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.

[Hin00c] 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, Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[Hin00d] Ralf Hinze. Polytypic values possess polykinded types. In R. Backhouse and J. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000*, July 2000.

[Hin01] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, September 2001.

[Hoo97] Paul Ferenc Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Eindhoven University of Technology, 1997.

[Jay95] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jeu01] Johan Jeuring. Generic haskell: a language for generic programming, 2001. Available from http://www.generic-haskell.org/.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP—A polytypic programming language extension. In *Conf. Record 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'97, Paris, France, 15–17 Jan 1997*, pages 470–482. ACM Press, New York, 1997.

[JL] S. Jones and J. Launchbury. Explicit quantification in haskell. Available from http://research.microsoft.com/Users/simonpj/Haskell/quantification.html.

[Jon97] Mark P. Jones. First-class polymorphism with type inference. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 15–17 1997.

[Mal89] G. Malcolm. Homomorphisms and promotability. In *J.L.A. van de Snepscheut, editor, Conference on the Mathematics of Program Construction*, number 375 in LNCS, pages 335–347. SpringerVerlag, 1989.

[Mar01] Clare Martin, 2001. Private communication.

[McC79] Nancy Jean McCracken. *An investigation of a programming language with a polymorphic type structure*. PhD thesis, Syracuse University, 1979.

[Mee96] Lambert Meertens. Calculate polytypically. In H. Kuchen and S. D. Swiestra, editors, *Proceedings 8th Intl. Symp. on Programming Languages: Implementations, Logics, and Programs,*

PLILP'96, Aachen, Germany, 24–27 Sept 1996, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, Berlin, 1996.

[MG01] Clare Martin and Jeremy Gibbons. On the semantics of nested datatypes. *Information Processing Letters*, 80:233–238, 2001.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *M. Paul and B. Robinet, editors, Proceedings of the International Symposium on Programming, number 167 in Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

[Oka98a] Chris Okasaki, 1998. Private communication.

[Oka98b] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[Oka99] Chris Okasaki. From fast exponentiation to square matrices: An adventure in types. In *International Conference on Functional Programming*, pages 28–35, 1999.

[Pat98] Ross Paterson, 1998. Private communication.

[Pat01] Ross Paterson. Short course on arrows and computation, 2001. Available from
http://www.soi.city.ac.uk/ ross/arrows/ShortCourse.html.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. In *R. E. A. Mason (ed.): Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France*, pages 513–523. Amsterdam North-Holland, 1983.

[Wad89] P. Wadler. Theorems for free! In *The 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359, London, September 1989. Imperial College, ACM Press.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[Xi99] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

[Zen97] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187(1–2):147–165, 1997.

Index

ω-cocontinuous hofunctors, 31
τ-Δ calculus, 123

accumulating parameters, 48
algebra family folds, 29
algebra family folds
  fusion laws of, 30
allegories, 82
alternating lists, 33, 59
alternating lists
  separation of, 35
auxiliary parameters, 70
AVL Trees, 6
axioms of allegories, 82
axioms of categories, 13

base functors, 21
bifunctors, 17
broadcast operations, 122
bushes, 65

categories, 5, 13
category of functions, 14
category of relations, 82
Cayenne, 11
characterisation of membership, 108
Church numerals, 8
composition of arrows, 14
constant functors, 25
converse operator, 82
converse operator
  axioms of, 82
  distributes over meet, 83
coproducts, 18
coproducts
  fusion laws of, 19
  universal property of, 18

de Bruijn terms, 56
Dependent ML, 9, 171
division, 101
division
  another law, 116
  cancellation law of, 101
  extra laws, 111

efficient fold operator
  for bushes, 80
  for nests, 49
  preserves injectivity, 102
efficient summation
  on nests, 50
embedding
  bushes, 144
  compositions, 155
  identities, 155
  initial algebras, 156
  map operations, 155
  nests, 142
  using zips, 143
  zips, 155
embedding function, 140

embedding operator, 146
embedding predicates, 140, 159
embedding target, 140
endofunctor category, 27
endofunctors, 16
endofunctors
    preserve composition, 16
    preserve identities, 16
endorelator category, 81
endorelators, 81
expressivity of nested datatypes, 6

fans, 114
fans
    fixpoint case, 116
    polynomial cases, 114
fixpoint equation, 15
flattening
    nests, 30
    nests
        as an efficient reduction, 55
    square matrices, 37
fold operators, 2
fold-equality law, 4
fold-equality law
    for nests, 53
fold-equivalence law, 149
fold-fusion law, 3
fold-fusion law
    for efficient folds, 104
    for efficient reductions
        generic version, 105
        on nests, 53
    for embedding operator, 148
    for generalised folds
        generic version, 76, 77
        on bushes, 76
        on nests, 46
        on nests (specialised), 47
    for reductions, 78
    for reductions
        on nests, 46
fork operator, 17
functors, 5
functors
    composition of, 25

generalised fold operator
    for alternating lists, 59
    for bushes, 67
    for nests, 40, 41
    generic version, 70
generalised folds, 4, 39
generalised folds
    vs. Hinze's folds, 68
Generic Haskell, 12
generic operations, 3

higher-order naturality of zip, 131
Hindley-Milner, 11
Hinze's recursive bag equations, 171
hofunctors, 28
horelators, 91
hylomorphism theorem, 100

identity arrows, 14
identity functor, 20
indexed types, 11
initial algebras, 22
initial algebras
    existence of, 31, 88
injective relations, 101
isomorphism, 15

join operator, 18

kinds, 8

Knaster-Tarski theorem, 97

Lambek's lemma, 22
last operation, 151
lax natural transformations, 87
lax natural transformations
  characterisation of, 108
  preserved by Λ, 90
linear datatypes, 4
locally complete allegories, 84

main parameter, 70
map operations, 20
map operations
  as efficient folds, 92
  as standard folds, 24
map operators, 2, 20
map-fusion law, 3
map-fusion law
  for efficient folds, 63, 80
  for generalised folds
    generic version, 74
    improved generic version, 75
    on bushes, 74
    on nests, 45
meet operator, 82
membership, 4, 106
membership
  binary relators, 110
  fixpoint case, 111
  polynomial cases, 109
  relator without, 108
membership relation, 106
modelling Haskell, 14
monotonicity
  of relators, 95
  of subset inclusion, 82
multirelators, 123
Mycroft-Milner, 11

natural transformations, 20
natural transformations
  typing rules of, 21
nested fixpoint cases, 32
newtype keyword, 42
non-linear datatypes, 4

partial order, 82
point-free style, 14
pointed complete partial orders, 14
pointwise style, 14
polyfunctorial, 106
polymorphic recursion, 11, 51
polynomial cases, 32
polynomial functors, 25
polynomial hofunctors, 32
PolyP, 12
postcomposition functor, 131
power allegories
  cancellation law of, 89
  fusion law of, 89
  universal property of, 89
power set endorelator, 89
power trees, 1
precomposition functor, 131
product category, 19
product functor, 17
products, 17
products
  universal property of, 17
program calculation, 3
projection functor, 26
projections, 17
proper natural transformations, 87
properly nested functors, 32

Quick Check, 169

rank two type signatures, 12, 29
reduction operator, 44
reductions, 44
refinement types, 8
regular endofunctors, 24
relational extensions, 85
relational product, 84
relational product
  absorption law of, 84
relations, 4
relators, 85
relators
  commute with converse, 85
  preserve total functions, 92
reversing
  a list (naively), 48
  a power tree, 155
  a tree, 155

sectioning, 18
shape, 130
shape behaviour of zip
  formally, 130
  informally, 120
shunting functions, 84
simple fold operator
  for alternating lists, 35
  for nests, 28
  for square matrices, 37
simple folds
  as generalised folds, 40
  vs. efficient reductions, 55
simple relations, 96
single case grammar for polynomial hofunctors, 69
source object, 13
square matrices, 7, 59
standard fold operator
  for lists, 22
  generic version of, 26
standard folds, 22
strict continuous functions, 14
substructures, 122
summation
  for lists, 23, 38
  generic version, 71
  Hinze's generic version, 73

tabular allegories, 95
tail-recursive datatypes, 152
target object, 13
terminal object, 16
total relations, 96
transpose operation, 122
trie data structures, 2
type constructor, 15

undecidability of type inference, 11
unfans
  for nests, 115
unfold, 100
union operator
  composition distributes over, 83
universal property, 3
universal property
  of efficient folds (proof), 61
  of generalised folds (proof), 61
  of meet, 82
unzip function in Haskell, 120

zip operations, 4, 120
zip operations
  binary relators, 125
  fixpoint case, 127
  polynomial cases, 124