Structured recursion for non-uniform data-types

by Paul Alexander Blampied, B.Sc.

Thesis submitted to the University of Nottingham

for the degree of Doctor of Philosophy, March 2000

Contents

Abstract

Recursion is a simple but powerful programming technique used extensively in functional languages. For a number of reasons, however, it is often desirable to use structured forms of recursion in programming, encapsulated in so-called recursion operators, in preference to unrestricted recursion. In particular, using recursion operators has proved invaluable for transforming and reasoning about functional programs. The recursion operators *map* and *fold* are well-understood for uniform data-types, and recent work on applications of non-uniform data-types has motivated extending these recursion operators to non-uniform data-types as well.

Previous work on defining recursion operators for non-uniform data-types has been hampered by expressibility problems caused by parametric polymorphism. In this thesis, we show how these problems can be solved by taking a new approach, based upon the idea of “algebra families”. We show how these recursion operators can be realized in the functional language Haskell, and present a categorical semantics that supports formal reasoning about programs on non-uniform data-types.

Acknowledgements

I would like firstly to thank my supervisors, Graham Hutton and Mark Jones, for steering me in the right direction and for always making the time to help me.

My gratitude goes to my colleagues in Lapland for putting up with my incessant ignorant questions: Tony Daniels, Ben Gaster, Claus Reinke, Colin Taylor and Richard Verhoeven, and all the other members, past and present, of the Languages and Programming group.

Special thanks to my family for supporting and encouraging me, and my friends for making my time in Nottingham much more than just an academic experience.

Finally, I would like to thank the States of Jersey Education Committee for supporting my research with a three-year grant, without which this thesis would not have been possible.

Chapter 1. Introduction

Data-types in programming languages allow us to organize values according to their properties and purpose. Basic data-types include integers, floating-point numbers, booleans, characters, strings, and so on. Most languages provide ways to construct more complex types such as arrays, lists and trees. The ability to express and use complex data-structures in a flexible and straightforward way is an important measure of the usability and expressiveness of a programming language. Modern functional languages excel at this, and their type systems are being continually developed to bring greater expressiveness and control to the programmer. In this thesis we concentrate on a class of data-types called *non-uniform* or *nested* [BirdM98] data-types that can be defined in the powerful polymorphic type systems of modern functional languages such as Haskell [PeytonJonesH99] and ML [MilnerTHM97], but have received little attention because the same type systems prevent the programmer from defining practically useful functions over them.

Haskell and ML have type systems based on the Hindley–Milner type system [Milner78]. This system allows all functions to be *statically* typed at compile-time, thus ensuring that no run-time type errors can occur. It allows the definition of *recursive* types which can describe data-structures of varying size. The most common example of a recursive data-type is that of lists, which are ubiquitous in functional programming.

In contrast to *uniform* recursive types such as lists and binary trees where the structure remains the same at different recursion levels, *non-uniform* or *nested* data-types have structures that *change* with recursion depth. The changing structure in these data-types makes it difficult or impossible to define useful functions over them using the Hindley–Milner type system, but some recent extensions to the

Haskell type system, namely *polymorphic recursion* [Mycroft84] and *rank-2 type signatures* [McCracken84, PeytonJones], have removed many of the technical barriers and such definitions are now feasible.

Researchers have begun to explore applications of non-uniform data-types. The most promising of these is that non-uniform data-types can be used to encode certain structural invariants on uniform data-types, which can then be checked and enforced automatically by the type system.

Because of the extra structural complexity inherent in non-uniform types, recursive functions on non-uniform data-types are more difficult to design than similar functions on uniform types. This thesis advocates the use of recursion operators to impose a structure on such function definitions and hide the complexities from the programmer. In particular, we transfer and generalize the recursion operators *map* and *fold* to operate on non-uniform data-types. These provide “prepackaged” recursion for a particular type, and are general enough to be suitable for many common operations.

Map and fold are central to the Squiggol [Bird87, Malcolm90, BirddM97] style of transformational programming for functional languages, and, for uniform types, have an elegant and well-understood semantics in category theory [Hagino87, Malcolm90, FokkingaM91]. As we transfer the map and fold operators to non-uniform data-types, we develop an appropriate categorical semantics that will allow us to prove certain properties and laws about them following in the spirit of Squiggol.

The following sections describe in more detail the class of non-uniform data-types we are considering and why they cause problems in functional programming. We also introduce the notions of *map* and *fold* through their role in Squiggol.

1.1. Non-uniform data-types

A *uniform* parameterized recursive data-type is one in which all the recursive calls are made at the same instance of the parameters, for example, the following Haskell definition of cons-lists:

```
data List a = NilL
            | ConsL a (List a)
```

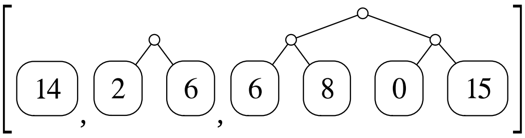
A *non-uniform* data-type is a parameterized recursive data-type in which one or more of the recursive calls are made at modified instances of the parameters. We take the terminology from Kubiak et al. [KubiakHL92] and Okasaki [Okasaki98, section 10.1] who call this “non-uniformly recursive” or just a “non-uniform” data-type. Compare the above definition of lists with the following non-uniform data-type:

```
type P a = (a,a)
data Nest a = NilN
            | ConsN a (Nest (P a))
```

The recursive call to Nest is made with the modified parameter (P a). We can see how the structure changes with recursion depth by looking at some example values of nests of integers:

```
NilN :: Nest Int
ConsN 3 NilN :: Nest Int
ConsN 8 (ConsN (6,12) NilN) :: Nest Int
ConsN 14 (ConsN (2,6) (ConsN ((6,8), (0,15)) NilN)) :: Nest Int
```

One way of thinking about nests is as lists of perfect binary leaf trees — the last example could be viewed as the list:



However, the non-uniform recursion in the nest data-type imposes the constraint that the first tree is just a single value and the trees increase exactly one level in depth at each position in the list.

It is possible to define data-types that combine uniform and non-uniform recursive calls, for instance:

```
data T a b = C1 a
           | C2 b (T a b) (T b a)
```

The first recursive call, (T a b) is uniform, but the second, (T b a), is non-uniform because the parameters have been switched. We can give even more pathological examples of non-uniform data-structures — one of the simplest is Bird and Meerten's [BirdM98] example:

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

Here the parameter to the recursive call is modified by the data-type being defined. Data-types such as this have been called "non-linear" non-uniform data-types, whereas the Nest data-type would be considered "linear" [BirdP99]

1.1.1. Applications of non-uniform types

Okasaki [Okasaki98, section 10.1.2] gives a good example of how non-uniform data-types can be used to enforce structural invariants. He gives the type of *sequences*, or *binary random-access lists*:

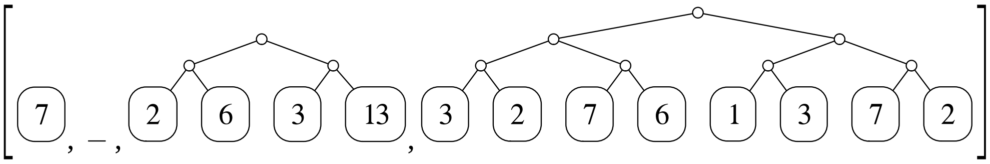
```
type P a = (a,a)
data Seq a = NilS
           | ZeroS (Seq (P a))
           | OneS a (Seq (P a))
```

Note that this is essentially just the Nest data-type extended with an extra constructor ZeroS that carries no values, but "nudges" the level of pairing up by one. What would be called the ConsN constructor for Nest has been given the name OneS to emphasize the correspondance with binary numbers. The number of elements in a sequence can be discovered by reading the constructors as a binary number — for example, the sequence:

```
OneS 7 (ZeroS (OneS ((2,6), (3,13)) (OneS (((3,2),(7,6)),((1,3),(7,2))) NilS))) :: Seq Int
```

which has thirteen elements. Thirteen is 1011 in binary (with least significant digit at the left), and this number can be read off from the constructors. Immediately this gives us an idea how to define an efficient length function for sequences, but what other benefits does using a non-uniform data-type bring us?

Just as nests, sequences can be thought of as lists of perfect binary leaf trees, but the additional ZeroS constructor gives us the ability to miss out trees in the list, and the above sequence of integers could be displayed as:



where " – " indicates the omitted tree caused by the ZeroS constructor.

The invariants are again fully captured by the non-uniform data-type, and this means that the invariants can be guaranteed by the type-checker. In particular, it means that *functions* that type-check and produce Seq values are guaranteed to enforce the invariant, that is, produce a valid sequence. As an example, consider consSeq — a function analogous to the constructor Cons on lists — that adds a new element to a sequence:

```
consSeq :: a → Seq a → Seq a
consSeq x NilS = OneS x NilS
consSeq x (ZeroS ps) = OneS x ps
consSeq x (OneS y ps) = ZeroS (consSeq (x,y) ps)
```

Cons-ing an element to a sequence is like adding one to a binary number — if the least significant digit is a zero then it becomes a one, otherwise, if it is a one, then we must perform a *carry* across to the next digit. The result of using consSeq to add an element to a well-formed sequence is another well-formed sequence, and this is proved automatically by the type-checker, with no extra effort from the programmer.

Using the sequence data-type, Okasaki develops functions for performing efficient look-up and update operations on sequences that run in logarithmic time.

The design of non-uniform data-types to capture particular structural invariants can be based on *number systems* [Okasaki98, Hinze99]. Different representations of the natural numbers correspond to data-structures that satisfy different invariants. Lists correspond to a *unary* representation of the natural numbers, whereas sequences correspond to a *binary* representation.

Okasaki [Okasaki98] gives several further examples of implementations of queues and deques (double-ended queues) that use non-uniform recursion and also develops data-types for square matrices [Okasaki99]. Hinze [Hinze99] develops non-uniform data-types that capture square matrices and rectangular matrices and several variants of trees such as Braun trees and 2–3 trees. A novel application is Bird and Paterson's [BirdP99] use of a non-uniform data-type to capture lambda terms in de Bruijn notation. The data-type's treatment of bound variables mean that the definitions of application and abstraction are particularly simple.

1.1.2. Why do non-uniform data-types cause problems in functional programming?

Anybody who has tried to write a function over a non-uniform data-type will know of the knots it is possible to get into with the type system. The type systems of

Haskell and ML are based on the Hindley-Milner type system [MilnerTHM97]. A peculiarity of this system is that, although non-uniform type definitions are perfectly legal, the type system makes it almost impossible to define interesting functions over non-uniform data-types. This is because recursive functions over non-uniform types generally require *polymorphic recursion* [Mycroft84]. Polymorphic recursion allows a polymorphic function to recursively call itself at a *different* polymorphic instance to the "parent" call. We have already seen an example of a function that uses polymorphic recursion — the consSeq function from section 1.1.1. The final line of the function definition,

```
consSeq x (OneS y ps) = ZeroS (consSeq (x,y) ps)
```

contains a recursive call to consSeq, but at a different instance.

The non-uniform recursion in non-uniform types means that the types of the elements in a data-value change with recursion depth — for instance, integers, pairs of integers, pairs of pairs of integers, and so on in the case of a nest or sequence of integers. A recursive function to process such data-values would have to deal with all the different element types, and this can be achieved by using polymorphic recursion to match the polymorphic function instance with the element types.

One problem with polymorphic recursion is that type inference is undecidable [Mycroft84]. Haskell (from version 1.3) supports functions that use polymorphic recursion, but the programmer must explicitly supply type signatures for such functions.

Another recent extension to the Haskell type system that we will make extensive use of is that of *rank-2 type signatures* [McCracken84, PeytonJones], which allow true polymorphic functions to be passed as arguments to other functions.

Even with these extensions to the type system, defining functions over non-uniform data-types is still not easy. The recursion in these functions will generally be more complex than similar functions over uniform types. This, compounded with the lack of type inference, makes writing functions over non-uniform data-types a tediously error-prone task. For this reason, provision for structuring recursive function definitions is arguably *more* important for non-uniform types than for uniform types, as it would remove or greatly reduce the possibility of type errors by providing simplified combinators to deal with the recursion.

1.1.3. Why do non-uniform data-types not fit into the categorical theory of data-types?

In chapter 2, we will review the basic categorical theory of data-types as initial algebras [Hagino87, Malcolm90]. The usual way in which this theory is used deals essentially with *non-parameterized* data-types such as the type of natural numbers or lists of a given fixed type. When faced with a parameterized type, the parameters are fixed and the resulting non-parameterized type is formed. This works for uniform parameterized types for which the recursive call is always made at the same parameters, but it is the nature of non-uniform types that their parameters are modified in the recursive call. It is therefore impossible to fix the parameters and consider the resulting non-parameterized type. We can illustrate this problem at the syntactic level in Haskell by comparing parameterized lists and nests. If we fix the parameter of the list type definition to some type, say integers, then we arrive at the non-parameterized type of integer lists:

```
data IntList = NilL | ConsL Int IntList
```

simply by replacing every occurrence of (List a) by IntList. If we try the same trick with nests, then we encounter a problem:

```
data IntNest = NilN | ConsN Int ?
```

With what do we replace (Nest (P a))? We cannot use IntNest, instead we need the type of nests of *pairs* of integers, let us call it PairIntNest. But this, in turn, depends on the definition of PairPairIntNest, and so on. The upshot is that we cannot define a single instance of a non-uniform type because that instance must be defined in terms of infinitely many other instances.

A large part of this thesis is concerned with transferring the theory surrounding initial algebras to alternative settings in which it can be applied successfully to non-uniform data-types.

1.2. Program calculation

Calculational or transformational programming is the process of transforming program specifications into (hopefully efficient) implementations using transformations that are guaranteed to preserve program correctness. The idea is that it is easier to state a simple and correct specification of a problem and then try to transform it into an efficient implementation rather than trying to produce an efficient and correct implementation from the beginning.

Because of their relatively simple mathematical foundations and their *referential transparency*, functional languages have achieved greater success with transformational programming calculi than have other languages. Calculi for functional programming languages are often algebraic in nature, and it is often possible to *calculate* a program as a "solution" to a specification in a manner analogous to problems of algebra in school:

$$
\begin{array}{ll}
 & x^2 - 6 = 3 \qquad \text{(specification)} \\
\equiv & \qquad \{\text{ add 6 to both sides }\} \\
 & x^2 = 9 \\
\equiv & \qquad \{\text{ square root }\} \\
 & x = \ \pm 3 \qquad \text{(solutions)}
\end{array}
$$

There have been several calculi developed for functional languages. Burstall and Darlington [BurstallD77] introduced "fold / unfold" transformations, although the terminology should not be confused with the different meaning of “fold” that we use in this thesis. Backus [Backus78] presented a functional programming language based on combinators, and gave algebraic laws that the combinators satisfied. Bird [Bird87] developed an algebraic theory of lists whose operators could easily be expressed in a functional language, and formed the basis for an effective calculus for transforming list-based programs. This calculus, and various extensions and variations of it, became known as the “Bird–Meertens” formalism or, more informally, “Squiggol”, because of the notation used. It is from Squiggol that we draw the concepts of map and fold.

1.3. Maps and folds in Squiggol

Two of the most common and important operators in the Squiggol program calculation style are *map* and *fold*. Both, in their original forms, operate on lists. The behaviour of both operators is described below:

Map The map operator takes a function from A to B as an argument and applies that function across a list of A’s to produce a list of B’s. The Squiggol notation for map is a postfix asterisk, so the operation that maps a function f across a list would be written as $f*$, and its behaviour is:

$$f*[x_1, x_2, \ldots] = [f(x_1), f(x_2), \ldots]$$

Fold Fold captures a more powerful processing pattern on lists than map. Given a list, it is common to want to combine the elements of the list in some way, for example, to sum a list of integers, or find the maximum value in the list. Fold generalizes operations like this by placing an arbitrary binary operator between the elements of the list. The binary operator is given as an argument to the fold. The fold must also know what value of the result

> type to return if it is applied to an empty list — this is given as another argument. Given a binary operator $\otimes$ and a default value e, the resulting fold is written as $\llparenthesis e, \otimes \rrparenthesis$ and its behaviour when applied to a list is
>
> $$\llparenthesis e, \otimes \rrparenthesis [x_1, x_2, \ldots, x_n] = x_1 \otimes (x_2 \otimes \cdots \otimes (x_n \otimes e) \cdots)$$
>
> If the binary operator is associative then there is no need for brackets in the result:
>
> $$x_1 \otimes x_2 \otimes \cdots \otimes x_n \otimes e$$
>
> Further, if e is a unit of $\otimes$ then the result simplifies to
>
> $$x_1 \otimes x_2 \otimes \cdots \otimes x_n$$
>
> for non-empty lists. Therefore, to sum a list of numbers we can use fold with addition as the associative binary operator and zero as the default value, which is a unit of addition. More complicated operations require more thought to be put into the choice of binary operator and default value. Squiggol notation for folds has become standard with the “banana bracket” notation: $\llparenthesis e, \otimes \rrparenthesis$, or just $\llparenthesis \otimes \rrparenthesis$ if the unit value is obvious or understood. In this thesis we will use the slight variation of $\llparenthesis e \triangledown \otimes \rrparenthesis$ for reasons that will become clear when we introduce categorical data-types.

Backus’ language FP [Backus78] has a map operation called “apply to all” and also an operation similar to fold called “insert”. Fold has its origins in the APL “reduction” operator [Iverson62]. Although these two patterns may seem somewhat limiting, it turns out that they are very powerful — most common list functions can be expressed using maps and folds. Hutton [Hutton99] gives more information on the expressive power of the fold operator. In fact, maps can even be expressed as folds.

It is often *desirable* to express functions in terms of maps and folds because

- the map and fold patterns of recursion are simple to understand and familiar to functional programmers;

- the programmer need only supply the ingredients to the map or fold — the recursion is taken care of behind the scenes; this results in quicker development time and less opportunity for programmer error;

- functions defined using map and fold are amenable to transformation and are also subject to useful theorems such as knowledge of their termination properties;

- some compilers for functional languages can automatically transform expressions involving folds and similar operators and thus eliminate intermediate data-structures; this is known as *deforestation* [TakanoM95].

Having said this, it is worth noting that not all functions on lists fit naturally into the fold recursion pattern, and certain functions can never be written as folds.

So far we have only considered transformations for programs based on lists. Two breakthroughs that have made these Squiggol forms more relevant to functional programming are:

- the discovery that operations analogous to map and fold exist for all *regular* data-types [Malcolm90];

- that the Squiggol theory, which was originally developed in the setting of sets and total functions, can be transported to the functional programming world of ω-complete pointed partial orders and (ω-continuous) partial functions [FokkingaM91], although some strictness conditions must be satisfied.

Thus the Squiggol theory is applicable to many of the data-types that can be defined in functional programming languages, including the infinite structures that are frequently used in lazy functional programming.

Squiggol forms such as map and fold can be implemented straightforwardly in functional languages and have become standard combinators in many functional languages. As a result, Squiggol specifications can be translated more-or-less directly into functional programs. For example, the Squiggol expression $(\![e, f]\!) \circ g*$ (where $\circ$ indicates function composition) would be written in Haskell as

```
(foldr f e) . (map g)
```

Conversely, functional programs written using combinators such as map and fold are subject to Squiggol-style transformations.

These Squiggol constructs have even influenced the design of new functional languages.

- Charity [CockettF92] is a polymorphic programming language that forgoes general recursion in favour of the structured recursion operators map and fold (and its dual, unfold) and strong data-types [CockettD92], and is based directly on the categorical semantics of these operators on initial and final data-types.
- "Polytypic" or "generic" programming [BackhouseJJM99] allows algorithms to be designed that are parameterized by the *structure* of data-types, and work uniformly over data-types that have differing structures, such as lists and trees. Applications include unification, pretty-printing, parsing and data compression. Fold and map are "polytypic" operations, and form the basis of many other polytypic operations. PolyP [JanssonJ97] is an extension of Haskell that provides facilities for defining polytypic functions.

1.4. Related work

Several authors have investigated folds for non-uniform data-types. Bird and Meertens [BirdM98] describe folds with polymorphic arguments (the "higher-order catamorphisms” in chapter 3 of this thesis). They note the significant limitations of these folds and try to overcome them by “reducing” the non-uniform structure to a uniform type, such as a list, and then folding over the resulting uniform type. This has the disadvantage that the extra information contained in the non-uniform structure is lost and cannot be exploited in the fold. In a later paper, Bird and Paterson [BirdP99] present *generalized folds* which provide a more elegant solution to the problem and perform a similar function to the folds described in chapter 4 of this thesis. Independently, Jones [Jones98] was also experimenting with folds with polymorphic arguments, and work with him [JonesB99] led to the “algebra family” approach described in chapter 4 of this thesis.

Hinze [Hinze99] has taken a rather different approach to defining functions over non-uniform data-types. Instead of trying to give them an initial algebra semantics, he uses *algebraic trees* derived from the data-type definitions as indexing sets for *polytypic* functions. In this way he can treat both uniform and non-uniform types within the same polytypic framework, although this does not yet support structured recursion schemes such as fold. Lacking the initiality property of folds, one must resort to other methods such as fixed-point induction to prove calculational laws for polytypic functions.

1.5. Purpose and contributions of this thesis

In this thesis we develop an approach to constructing recursion operators analogous to map and fold for non-uniform data-types. The fold operators are general enough to be suitable for many operations on non-uniform data-types and do not suffer the limitations of the “higher-order folds” described by Bird and Meertens [BirdM98].

We give our fold operators categorical semantics in terms of initial algebras and give sufficient conditions for the existence of the initial algebras for particular categories, notably the category of sets and total functions and the category of ω-complete pointed partial orders and ω-continuous functions, thereby justifying the calculational laws that stem from initiality.

It should be emphasized that the approach to defining functions over non-uniform data-types we describe in this thesis is oriented towards the practicality of doing so in *currently available* extensions to the type systems of popular functional languages. This means that we are still restricted to work within the bounds of the type system.

Chapter 2 reviews the categorical treatment of data-types as initial algebras and folds as catamorphisms. Chapter 2 also covers some of the technical machinery that can be used to prove the existence of initial algebras in various categories.

In chapter 3 we lift the categorical theory to *functor categories* and show how "higher-order catamorphisms" translate into Haskell fold operators which take polymorphic arguments. We discuss the expressive limitations of catamorphisms in functor categories and the problem of proving the existence of initial algebras. The main contribution of this chapter is the lifting of results for proving the existence of initial algebras to functor categories.

In chapter 4 we develop a different approach to folding over non-uniform types that overcomes the expressive limitations of catamorphisms in functor categories by supplying *families* of functions as arguments to the fold operators instead of polymorphic functions. We call the argument families *algebra families* and the fold operators *algebra family folds*. We show how to define these fold operators first of all in Haskell, and then in chapter 5 we capture the recursion pattern categorically as catamorphisms in particular functor categories. We adapt and extend our results from chapter 3 about the existence of initial algebras in functor categories to apply to algebra family folds.

Chapter 6 uses algebra family folds to define map operators for non-uniform types and we prove the calculational properties of map by calling on the calculational properties of algebra family folds.

The concluding chapter 7 summarizes the thesis and outlines possible future work.

Chapter 2. Categorical data-types

Category theory has been successful in modelling many of the constructions that arise in theoretical computer science, particularly in the λ-calculus and functional programming. Squiggol fits naturally within a categorical framework — Malcolm [Malcolm90] pioneered this approach, building on foundational work of Hagino [Hagino87]. Fokkinga [Fokkinga92, Fokkinga92] and Bird and de Moor [BirddM97] provide tutorials on the categorical calculation style.

In this thesis we are concerned with the categorical interpretation of data-types as *initial algebras*, which leads naturally to the fold pattern of structured recursion. Unfortunately, non-uniform data-types do not fit into the usual framework, and we are forced to look for other approaches to defining folds for them. This chapter reviews the traditional construction of data-types as initial algebras, and later, in chapters 3 and 5, we will see how the same construction can be used in different categories to provide folds for non-uniform data-types.

A modest knowledge of category theory is required for the theoretical parts of this thesis — Pierce's book [Pierce91] and that of Barr and Wells [Barr90] cover the basics for the computer scientist, and the standard reference text is that of Mac Lane [MacLane71]. The basic definitions that we use are collected in appendix B for convenient reference. Explicitly, the reader should be familiar with initial objects, sums and products, and (co)cones and (co)limits. Functor categories play an important role in this thesis, and a good understanding of functors and natural transformations is necessary. Familiarity with the construction of categorical data-types would certainly be beneficial, although we review the main points here.

2.1. Notation

We first of all describe some of the categorical notation used throughout this thesis. We have tried to adhere to standard notation where it exists.

Composition of arrows is written $g \circ f$, although if we wish to emphasize or clarify the types of the arrows involved, we will display such a composition as

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Sums and *products* are denoted standardly as + and ×, with the associated binary operations *join* and *split* written respectively as $\triangledown$ and $\vartriangle$. Join can be viewed as a case construct — given two arrows $f : A \to C$ and $g : B \to C$, their join has type $f \triangledown g : A + B \to C$. Split is used to construct pairs — given two arrows $f : A \to B$ and $g : A \to C$ then their split has type $f \vartriangle g : A \to B \times C$. Left and right *injections* into a sum are written as $\grave{\iota}$ and $\acute{\iota}$, and *projections* from a product as $\grave{\pi}$ and $\acute{\pi}$.

Functors are essentially mappings of arrows and we display them as such:

$$\begin{tikzcd}[row sep=small]
F : \mathbb{C} \arrow[r] & \mathbb{D} \\
A \arrow[d, "f"'] \arrow[dr, phantom, "\mapsto"] & F.A \arrow[d, "F.f"] \\
B & F.B
\end{tikzcd}$$

although we will often write them more compactly using *sectioning* and *lifting* defined below. Both functor composition and functor application are traditionally indicated by juxtaposition, and it is usually apparent which is meant from the context. However, when we move to functor categories in chapters 3 and 5, the distinction is no longer obvious, and to avoid confusion we use an infix period to show functor application, for example $F.A$, reserving juxtaposition for composition, as in GF. We also allow functor composition to bind more tightly than application, and application to associate to the right so that, for example,

$(HGF).A$

$=$ { composition binds tighter than application }

$HGF.A$

$=$ { functor composition }

$HG.(F.A)$

$=$ { functor composition }

$H.(G.(F.A))$

$=$ { application associates to the right }

$H.G.F.A$

Infix bifunctors can be *sectioned*, which means fixing one of the arguments. Given a bifunctor $\otimes : \mathbb{C} \times \mathbb{D} \to \mathbb{E}$ and objects A in $\mathbb{C}$ or B in $\mathbb{D}$ then by $(A\otimes)$ and $(\otimes B)$ we mean respectively the functors

$$\begin{tikzcd}[row sep=small] (A\otimes) : \mathbb{D} \arrow[rr] & & \mathbb{E} \\ X \arrow[d, "f"'] \arrow[rr, phantom, "\mapsto", yshift=-3ex] & & A \otimes X \arrow[d, "Id_A \otimes f"] \\ Y & & A \otimes Y \end{tikzcd} \qquad \begin{tikzcd}[row sep=small] (\otimes B) : \mathbb{C} \arrow[rr] & & \mathbb{E} \\ X \arrow[d, "f"'] \arrow[rr, phantom, "\mapsto", yshift=-3ex] & & X \otimes B \arrow[d, "f \otimes Id_B"] \\ Y & & Y \otimes B \end{tikzcd}$$

Bifunctors can be *lifted* to operate on functors in the following way. For a bifunctor $\otimes : \mathbb{C} \times \mathbb{D} \to \mathbb{E}$ and functors $F : \mathbb{B} \to \mathbb{C}$ and $G : \mathbb{B} \to \mathbb{D}$ then we define

$$\begin{tikzcd}[row sep=small] F \dot{\otimes} G : \mathbb{B} \arrow[rr] & & \mathbb{E} \\ A \arrow[d, "f"'] \arrow[rr, phantom, "\mapsto", yshift=-3ex] & & F.A \otimes G.A \arrow[d, "F.f \otimes G.f"] \\ B & & F.B \otimes G.B \end{tikzcd}$$

Lifted functors play an important role in this thesis and we give a more comprehensive description of functor lifting in section 3.1.

For natural transformations we place the dot *below* the arrow to avoid any confusion with lifted functors and so that we can easily display composite natural transformations:

$$F \xrightarrow[\bullet]{\alpha} G \xrightarrow[\bullet]{\beta} H$$

For any object A in a category $\mathbb{D}$, we write the corresponding constant functor from another category $\mathbb{C}$ as $\underline{A}$:

$$\begin{tikzcd}
\underline{A} : \mathbb{C} \arrow[rr] & & \mathbb{D} \\
X \arrow[d, "f"'] & \mapsto & A \arrow[d, "Id_A"] \\
Y & & A
\end{tikzcd}$$

We introduce the following notation for cones and cocones. A functor $D : \mathbb{J} \to \mathbb{C}$ is a $\mathbb{J}$-diagram in $\mathbb{C}$. A cone over such a diagram is a natural transformation $\alpha : \underline{A} \underset{\cdot}{\to} D$ for some object A in $\mathbb{C}$, which is the apex of the cone. We use the shorthand and hopefully more suggestive notation $\alpha : A \lhd D$. Dually we write $\beta : D \rhd B$ for a cocone.

2.2. Data-types as initial algebras

The notion of *algebra* is central to the definition of data-types.

Definition 2.1 (F-algebra): For an endofunctor $F : \mathbb{C} \to \mathbb{C}$, an *F-algebra* is a pair (A, α) where

(i) A is an object in $\mathbb{C}$, called the *carrier* of the algebra;

(ii) α is an arrow $F.A \to A$ in $\mathbb{C}$.

Note that this definition is slightly different from a category theorist's, who would also require that F is a *monad* [MacLane71, section VI.2], but this requirement is unnecessary for our purposes. It is common to give just the arrow component of an algebra — the carrier object can easily be discovered from the type of the arrow. We can give some example algebras: if F is the constant functor $\underline{\mathbb{1}}$ whose result is always the terminal object, then any *global element* $a : \mathbb{1} \to A$ of a type A is a $\underline{\mathbb{1}}$-algebra. If

F is the pairing functor $Id_{\mathbb{C}} \dot{\times} Id_{\mathbb{C}}$, then any binary operation $\otimes : A \times A \to A$ on a type A is a $Id_{\mathbb{C}} \dot{\times} Id_{\mathbb{C}}$-algebra. We will often join algebras together using sums — for example, the two algebras above could be joined together to form a new algebra:

$$\begin{tikzcd}
\mathbb{1} \arrow[r, "\grave{\iota}"] \arrow[dr, "a"'] & \mathbb{1} + A \times A \arrow[d, "a \triangledown \otimes"] & A \times A \arrow[l, "\acute{\iota}"'] \arrow[dl, "\otimes"] \\
 & A &
\end{tikzcd}$$

Then $a \triangledown \otimes : \mathbb{1} + A \times A \to A$ is a $(\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} Id_{\mathbb{C}})$-algebra. Note that we used *lifting* to write the functors for the algebras more concisely. It is easy to check that

$$\begin{aligned}
& \mathbb{1} + A \times A \\
= & \quad \{ \text{constant functor, lifted product functor, identity functor} \} \\
& \underline{\mathbb{1}}.A + (Id_{\mathbb{C}} \dot{\times} Id_{\mathbb{C}}).A \\
= & \quad \{ \text{lifted sum functor} \} \\
& (\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} Id_{\mathbb{C}}).A
\end{aligned}$$

We can also define structure-preserving maps between algebras:

Definition 2.2 (F-algebra homomorphism): For an endofunctor $F : \mathbb{C} \to \mathbb{C}$, an *$F$-algebra homomorphism* from an F-algebra (A, α) to another (B, β) is an arrow $h : A \to B$ in $\mathbb{C}$ such that

$$\begin{tikzcd}
F.A \arrow[r, "F.h"] \arrow[d, "\alpha"'] & F.B \arrow[d, "\beta"] \\
A \arrow[r, "h"'] & B
\end{tikzcd}$$

Then, for a particular endofunctor $F : \mathbb{C} \to \mathbb{C}$, we can construct a new category $Alg(F)$ whose objects are F-algebras and arrows are F-algebra homomorphisms. Composition and identities are inherited from the base category $\mathbb{C}$.

Suppose the category $Alg(F)$ has an initial object, let us call it $(\mu F, in^F)$, then μF is an object in $\mathbb{C}$ and $in^F : F.\mu F \to \mu F$ is an arrow in $\mathbb{C}$. We call $(\mu F, in^F)$ an *initial algebra* of F. Being initial means that for any F-algebra $\alpha : F.A \to A$, there is exactly one F-algebra homomorphism from in^F to α. The notation for this unique F-algebra homomorphism is $\llparenthesis \alpha \rrparenthesis^F$ and it is a $\mathbb{C}$-arrow from μF to A that uniquely satisfies

$$\begin{tikzcd}
F.\mu F \arrow[r, "F.\llparenthesis \alpha \rrparenthesis^F"] \arrow[d, "in^F"'] & F.A \arrow[d, "\alpha"] \\
\mu F \arrow[r, dotted, "\llparenthesis \alpha \rrparenthesis^F"'] & A
\end{tikzcd}$$

which is equivalent to the equation

$$\llparenthesis \alpha \rrparenthesis^F \circ in^F = \alpha \circ F.\llparenthesis \alpha \rrparenthesis^F \tag{2.2.1}$$

The F superscript will usually be omitted if it is obvious from the context. The $\llparenthesis \alpha \rrparenthesis$ arrow is dotted to indicate its uniqueness. These unique homomorphisms are called *catamorphisms*, and we will see that they generalize the notion of folds on lists to arbitrary data-types.

An important fact about initial algebras is that they are isomorphisms:

> **Lemma 2.1 (Lambek's lemma):** If $F : \mathbb{C} \to \mathbb{C}$ has an initial algebra $(\mu F, in^F)$, then in^F is an isomorphism with inverse $\llparenthesis F.in^F \rrparenthesis : \mu F \to F.\mu F$.

This allows us to transform equation 2.2.1 above to give a recursive definition of $\llparenthesis \alpha \rrparenthesis$:

$$\llparenthesis \alpha \rrparenthesis = \alpha \circ F.\llparenthesis \alpha \rrparenthesis \circ \left(in^F \right)^{-1} \tag{2.2.2}$$

Let's see what this means for particular functors — the standard example is to construct the natural numbers as an initial algebra. Let $\mathbb{C}$ be a category with a terminal object $\mathbb{1}$ and binary sums, then define the functor F to be

$$\begin{tikzcd}[row sep=small] F : \mathbb{C} \arrow[r, "\longrightarrow", phantom] & \mathbb{C} \\ A \arrow[d, "f"'] \arrow[r, phantom, "\mapsto" description, shift right=3ex] & \mathbb{1} + A \arrow[d, "Id + f"] \\ B & \mathbb{1} + B \end{tikzcd}$$

We can give this definition more succinctly by lifting the sum functor:

$$F \overset{def}{=} \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}}$$

To see that this gives the same functor definition we can calculate

$$\begin{array}{ll} & (\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}}).A \\ = & \qquad \{ \text{ definition of } \dot{+} \} \\ & \underline{\mathbb{1}}.A + Id_{\mathbb{C}}.A \\ = & \qquad \{ \text{ constant functor and identity functor } \} \\ & \mathbb{1} + A \end{array}$$

for objects and

$$\begin{array}{ll} & (\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}}).f \\ = & \qquad \{ \text{ definition of } \dot{+} \} \\ & \underline{\mathbb{1}}.f + Id_{\mathbb{C}}.f \\ = & \qquad \{ \text{ constant functor and identity functor } \} \\ & Id_{\mathbb{1}} + f \end{array}$$

for arrows.

Suppose F has an initial algebra $(\mu F, in^F)$. Then in^F has type $\mathbb{1} + \mu F \to \mu F$. Any arrow from a sum can be written as the join of two arrows from the left and right components of the sum, let us call them Zero and Succ; they are defined as:

$$\mathsf{Zero} \overset{def}{=} \mathbb{1} \xrightarrow{\grave{\iota}} \mathbb{1} + \mu F \xrightarrow{in^F} \mu F$$

$$\mathsf{Succ} \overset{def}{=} \mu F \xrightarrow{\acute{\iota}} \mathbb{1} + \mu F \xrightarrow{in^F} \mu F$$

It is easy to show that $in^F = \mathsf{Zero} \triangledown \mathsf{Succ}$. The arrows Zero and Succ are called the *constructors* of the data-type μF because they allow values of type μF to be constructed, for example,

$$\mathbb{1} \xrightarrow{\mathsf{Zero}} \mu F$$

$$\mathbb{1} \xrightarrow{\mathsf{Zero}} \mu F \xrightarrow{\mathsf{Succ}} \mu F$$

$$\mathbb{1} \xrightarrow{\mathsf{Zero}} \mu F \xrightarrow{\mathsf{Succ}} \mu F \xrightarrow{\mathsf{Succ}} \mu F$$

and so on. Furthermore, because in^F is an isomorphism, we can recover the sequence of constructors that were used to build a particular value of μF. It should be clear that this allows us to encode and decode the natural numbers — the above examples encode the natural numbers 0, 1 and 2 respectively.

To understand the function of catamorphisms, it is more instructive to consider the construction of a more interesting type — lists of some arbitrary fixed type, say integers. Assuming the type of integers is an object in our base category, then lists of integers can be constructed as the initial algebra of the base functor

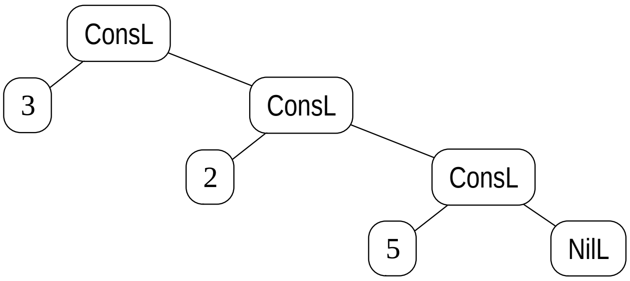
$$\begin{tikzcd} L : \mathbb{C} \arrow[r] & \mathbb{C} \\ X \arrow[d, "f"'] \arrow[r, phantom, "\mapsto"] & \mathbb{1} + Int \times X \arrow[d, "Id + Id \times f"] \\ Y & \mathbb{1} + Int \times Y \end{tikzcd} \tag{2.2.3}$$

or equivalently $L \overset{def}{=} \underline{\mathbb{1}} \dot{+} \underline{Int} \dot{\times} Id_{\mathbb{C}}$. Assuming an initial algebra $(List, in^L)$ exists, with $in^L : \mathbb{1} + Int \times List$ being the join of the two constructors

$$\mathsf{NilL} : \mathbb{1} \rightarrow List$$

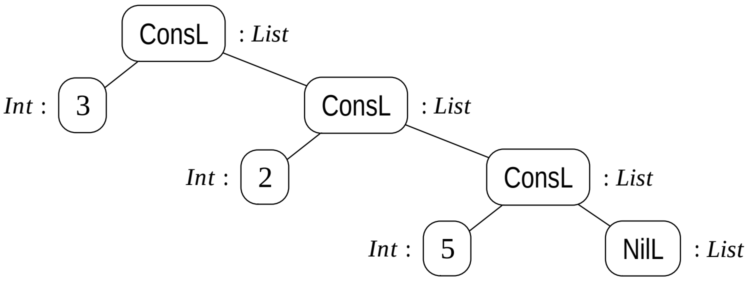
$$\mathsf{ConsL} : Int \times List \rightarrow List$$

Then *List* is an object in $\mathbb{C}$ and is the type of integer lists. A helpful way to visualize an integer list built using these constructors is as a tree diagram



This corresponds to the list [3, 2, 5]. The constructors are nodes in the tree diagram. ConsL has two arguments — one an integer and one a list of integers — and these are given as the two subtrees of the node. Constants such as integer values or the NilL constructor take no arguments and therefore appear at the leaves of the tree.

When we begin to talk about non-uniform data-types it will be helpful to decorate such diagrams with type information. The above tree would be decorated



(2.2.4)

To understand how catamorphisms operate, we can observe their action on the constructors of the data-type. Let $\alpha : \mathbb{1} + Int \times A \to A$ be an arbitrary L-algebra. We can think of α as the join of two arrows $\lambda : \mathbb{1} \to A$ and $\rho : Int \times A \to A$, so $\alpha = \lambda \triangledown \rho$. Consider then the catamorphism $(\!|\, \lambda \triangledown \rho \,|\!) : List \to A$; the catamorphism diagram is

$$\begin{tikzcd}
\mathbb{1} + Int \times List \arrow[r, "{Id + Id \times \llparenthesis \lambda \triangledown \rho \rrparenthesis}"] \arrow[d, "{in^L = \mathsf{NilL} \triangledown \mathsf{ConsL}}"'] & \mathbb{1} + Int \times A \arrow[d, "{\alpha = \lambda \triangledown \rho}"] \\
List \arrow[r, dotted, "{\llparenthesis \lambda \triangledown \rho \rrparenthesis}"'] & A
\end{tikzcd}$$

and is just a concise way of combining the two diagrams:

$$\begin{tikzcd}
\mathbb{1} \arrow[r, "Id"] \arrow[d, "\mathsf{NilL}"'] & \mathbb{1} \arrow[d, "\lambda"] \\
List \arrow[r, "{\llparenthesis \lambda \triangledown \rho \rrparenthesis}"'] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
Int \times List \arrow[r, "{Id \times \llparenthesis \lambda \triangledown \rho \rrparenthesis}"] \arrow[d, "\mathsf{ConsL}"'] & Int \times A \arrow[d, "\rho"] \\
List \arrow[r, "{\llparenthesis \lambda \triangledown \rho \rrparenthesis}"'] & A
\end{tikzcd}$$

into a single diagram. The two simpler diagrams correspond to the equations:

$$\llparenthesis \lambda \triangledown \rho \rrparenthesis \circ \mathsf{NilL} = \lambda \circ Id = \lambda$$

and

$$\llparenthesis \lambda \triangledown \rho \rrparenthesis \circ \mathsf{ConsL} = \rho \circ (Id \times \llparenthesis \lambda \triangledown \rho \rrparenthesis)$$

Thus $\llparenthesis \lambda \triangledown \rho \rrparenthesis$ replaces the NilL constructor by λ and the ConsL constructor by ρ while recursively applying $\llparenthesis \lambda \triangledown \rho \rrparenthesis$ to the *List* value that ConsL was applied to.

Let us construct a catamorphism on a list of integers. We first need an L-algebra $\alpha : \mathbb{1} + Int \times A \rightarrow A$ for some carrier type A. Let A be the type *Int*, then we can construct α as the join of two arrows with types

$$\mathbb{1} \rightarrow Int \qquad\qquad Int \times Int \rightarrow Int$$

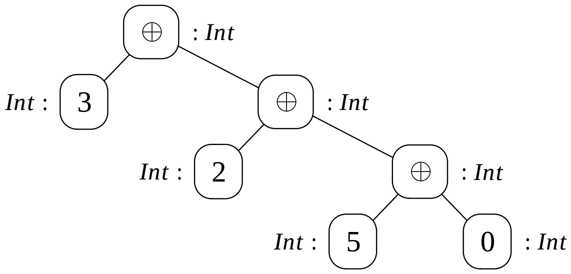
that is, an integer constant and a binary operation on integers. Some obvious candidates for these are zero and integer addition, and we get an L-algebra:

$$0 \triangledown \oplus : \mathbb{1} + Int \times Int \rightarrow Int$$

(writing integer addition as $\oplus$ to avoid confusion with categorical sums). Then, because in^L is initial, we know that there is a unique L-algebra homomorphism from in^L to $(0 \triangledown \oplus)$, and we call it $(\!|0 \triangledown \oplus|\!)$:

$$\begin{tikzcd}
\mathbb{1} + Int \times List \arrow[r, "Id + Id \times (\!|0 \triangledown \oplus|\!)"] \arrow[d, "\mathsf{NilL} \triangledown \mathsf{ConsL}"'] & \mathbb{1} + Int \times Int \arrow[d, "0 \triangledown \oplus"] \\
List \arrow[r, dotted, "(\!|0 \triangledown \oplus|\!)"'] & Int
\end{tikzcd}$$

Furthermore, we know the behaviour of $(\!|0 \triangledown \oplus|\!)$ — it replaces the NilL constructor with 0 and the ConsL constructor with $\oplus$ while recursively processing the second argument of the ConsL constructor. The effect of applying $(\!|0 \triangledown \oplus|\!)$ to the list in diagram 2.2.4 can be easily seen visually:



Notice how the types of the constructor nodes have changed to reflect the carrier of the new algebra. The result evaluates to the integer 10, which is the sum of the list. The effect of applying $(\!|0 \triangledown \oplus|\!)$ is to sum lists of arbitrary length.

Catamorphisms capture exactly the recursive processing pattern of Squiggol's folds on lists. Furthermore, catamorphisms generalize this pattern to other data-types that can be constructed as initial algebras. For these reasons, the terms 'fold' and 'catamorphism' (sometimes abbreviated to just 'cata') are often used interchangeably. In this thesis we will use 'catamorphism' in a strictly technical sense to mean the unique algebra homomorphism from an initial algebra to some other. By 'fold' we will mean not only catamorphisms but also the broader family of variations and generalizations that have been devised which follow the same basic pattern, but are not strictly catamorphisms.

The point we stress in this thesis is that *folds replace constructors*, and this is the intuition that we use for devising folds for non-uniform data-types.

2.3. Parameterized data-types

So far we have only looked at constructing single types such as the natural numbers or lists of integers. However, much of the construction of lists of integers is in common with lists of other types, for example, lists of booleans, or even lists of *lists* of integers. We would like to capture this similarity of construction of list types categorically, and be able to uniformly construct lists of arbitrary given types. The standard way of doing this is by using a base *bifunctor* rather than a unary base functor. The extra argument to the functor is to accept the *parameter* of the type. We give as an example the construction of parameterized lists. Recall the base functor for lists of integers:

$$\begin{tikzcd}
L : \mathbb{C} \arrow[r] & \mathbb{C} \\
X \arrow[d, "f"'] \arrow[r, phantom, "\mapsto"] & \mathbb{1} + Int \times X \arrow[d, "Id + Id \times f"] \\
Y & \mathbb{1} + Int \times Y
\end{tikzcd}$$

To construct lists of an arbitrary type, we abstract out any reference to the *Int* type and introduce the parameter type through an extra argument. Thus we get a bifunctor:

$$\begin{tikzcd}
\ddagger : \mathbb{C} \times \mathbb{C} \arrow[r] & \mathbb{C} \\
(A, X) \arrow[d, "(\alpha, f)"'] \arrow[r, phantom, "\mapsto"] & \mathbb{1} + A \times X \arrow[d, "Id + \alpha \times f"] \\
(B, Y) & \mathbb{1} + B \times Y
\end{tikzcd}$$

Notice that if we section ‡ with the *Int* type then we have

$$(Int\ddagger) = L$$

and it follows that the initial algebra of $(Int\ddagger) : \mathbb{C} \to \mathbb{C}$ gives the type of lists of integers. Furthermore, if we consider the initial algebra of $(A\ddagger) : \mathbb{C} \to \mathbb{C}$ for any type A in $\mathbb{C}$, then the carrier $\mu(A\ddagger)$ will be the type of lists of A, with the constructors:

$$\mathsf{NilL}_A : \mathbb{1} \to \mu(A\ddagger) \qquad \mathsf{ConsL}_A : A \times \mu(A\ddagger) \to \mu(A\ddagger)$$

forming the initial algebra.

By the same method, we can abstract the parameter out of any *uniform* parameterized type. For example, the bifunctor for binary trees would be:

$$\begin{tikzcd}
\dagger : \mathbb{C} \times \mathbb{C} \arrow[r] & \mathbb{C} \\
(A, X) \arrow[d, "{(\alpha, f)}"'] \arrow[r, phantom, shift right=4ex, "\mapsto"] & \mathbb{1} + A \times X \times X \arrow[d, "Id + \alpha \times f \times f"] \\
(B, Y) & \mathbb{1} + B \times Y \times Y
\end{tikzcd}$$

2.3.1. Maps can be expressed as folds

Catamorphisms or folds on parameterized types are expressive enough to capture maps as well. In this section we show how maps can be expressed as folds, and how this leads to *type functors*.

We demonstrate the technique, first of all, for parameterized lists. Suppose we have a function $f : A \to B$, and we would like to map it across a list with elements of type A to produce a list with elements of type B. That is, we would like to construct a function with type $\mu(A\ddagger) \to \mu(B\ddagger)$. We can construct this function as a catamorphism, but we must find the right algebra to give as the argument to the catamorphism operator. We want to produce a list of B's, so the carrier of the algebra must be $\mu(B\ddagger)$ — the type of lists of B-elements. So we need an $(A\ddagger)$-algebra with type

$$\alpha : A \ddagger \mu(B\ddagger) \to \mu(B\ddagger)$$

The algebra must be an $(A\ddagger)$-algebra because we are going to fold over the

$\mu(A\ddagger)$-structure. The behaviour of a map is that it applies the function f to every A-value in the list, but leaves the actual structure of the list unchanged. We can capture this in the following $(A\ddagger)$-algebra:

$$A \ddagger \mu(B\ddagger) \xrightarrow{f \ddagger Id} B \ddagger \mu(B\ddagger) \xrightarrow{in^{B\ddagger}} \mu(B\ddagger)$$

The role here of $in^{B\ddagger}$ is to replace the constructors at type A with the same constructors at type B, that is, NilL_A becomes NilL_B and ConsL_A becomes ConsL_B, although the order of the constructors, which determines the structure of the list, is unchanged. Thus we have an $(A\ddagger)$-algebra, and we use it to construct the catamorphism:

$$\begin{tikzcd}
A \ddagger \mu(A\ddagger) \arrow[r, "Id \ddagger \llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger}"] \arrow[dd, "in^{A\ddagger}"'] & A \ddagger \mu(B\ddagger) \arrow[d, "f \ddagger Id"] \\
 & B \ddagger \mu(B\ddagger) \arrow[d, "in^{B\ddagger}"] \\
\mu(A\ddagger) \arrow[r, dotted, "\llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger}"'] & \mu(B\ddagger)
\end{tikzcd}$$

We superscript the catamorphisms with $A\ddagger$ to emphasize that we are folding over the $\mu(A\ddagger)$-structure.

To see that this gives us the map function that we would expect, examine the behaviour of the catamorphism on the list constructors — first the NilL constructor:

$$\llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger} \circ \mathsf{NilL}_A$$

$=$ { catamorphism diagram }

$$\mathsf{NilL}_B$$

Then the ConsL constructor:

$$\llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger} \circ \mathsf{ConsL}_A$$

$=$ { catamorphism diagram }

$$\mathsf{ConsL}_B \circ (f \times Id) \circ (Id \times \llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger})$$

$=$ { composition of products, identity arrows }

$$\mathsf{ConsL}_B \circ (f \times \llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger})$$

Translating this into a Haskell definition, and abstracting out the function f as an argument, we get the standard definition of map on lists:

```
mapList :: (a → b) → (List a → List b)
mapList f NilL = NilL
mapList f (ConsL x xs) = ConsL (f x) (mapList f xs)
```

where (mapList f) plays the role of $\llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger}$.

Maps are constructed in an identical way for other parameterized types that can be captured using bifunctors.

2.3.2. Type functors

Now that we can define maps on parameterized types as folds, we can define *type functors*:

Definition 2.3 (type functor): Let $\ddagger : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ be a bifunctor such that for every A in $\mathbb{C}$, the endofunctor $(A\ddagger) : \mathbb{C} \to \mathbb{C}$ has an initial algebra $(\mu(A\ddagger), in^{A\ddagger})$. Then we define the *type functor* of $\ddagger$, denoted $\textcircled{\ddagger}$, to be:

$$\begin{tikzcd}
\textcircled{\ddagger} : \mathbb{C} \arrow[r] & \mathbb{C} \\
A \arrow[dd, "f"'] & \mu(A\ddagger) \arrow[dd, "\llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis^{A\ddagger}"] \\
{} \arrow[r, phantom, "\mapsto"] & {} \\
B & \mu(B\ddagger)
\end{tikzcd}$$

If we take $\ddagger$ to be the bifunctor for lists then the action of $\textcircled{\ddagger}$ on a type A in $\mathbb{C}$ is to transform it to the type $\mu(A\ddagger)$, that is, lists of A elements. In this respect, $\textcircled{\ddagger}$ behaves like the list *type constructor* in Haskell. The action of ⓣ on arrows is to map them across list structures.

2.3.3. Polynomial and regular data-types

The class of parameterized data-types that are usually considered in this categorical framework are called *regular* data-types, and consist of those derived from bifunctors built out of constants (for example $\mathbb{1}$ and *Int*), identity functors, products, sums and the type functors of other regular types. *Polynomial* types are derived from bifunctors built using constants, identities, products and sums — that is, everything that can be used in a regular type except for type functors. Simple extensions allow multiple-parameter and mutually-recursive types to be treated as well, and thus regular types account for a good proportion of the types that can be defined in modern functional languages. Types involving function spaces are non-regular, but it is possible to define folds over those as well [MeijerH95, FegarasS95].

An important fact about regular types is that they are all *uniform.* Recall that in order to construct lists of a given type A we

(i) first *fixed* the type A to get the sectioned functor $(A\ddagger){:}\mathbb{C} \to \mathbb{C}$,

(ii) then took the initial algebra of the functor $(A\ddagger)$ as our type.

In step (ii), the parameter A is fixed and cannot change. This corresponds to the property of uniform data-types that the parameter remains the same in any recursive calls in the data-type definition.

2.4. Calculation properties of catamorphisms

The calculational properties of catamorphisms arise from the fact that they are the *unique* algebra homomorphisms from an initial algebra. The simplest way that this uniqueness gives rise to a calculational law is that if h is an algebra homomorphism from an initial algebra then it is a catamorphism, that is, *the* unique algebra homomorphism:

Lemma 2.2: Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor with initial algebra $(\mu F, in^F)$, and $\alpha : F.A \to A$ any F-algebra. Then if h is an F-algebra homomorphism from in^F to α, then $h = (\!|\alpha|\!)$.

Proof: Immediate from the uniqueness of catamorphisms.

Written calculationally, this lemma gives us the rule:

$$h \circ in^F = \alpha \circ F.h \Rightarrow h = (\!|\alpha|\!)$$

The left-hand side is just the property that h is an F-algebra homomorphism. Clearly the reverse implication is also true because $(\!|\alpha|\!)$ is an F-algebra homomorphism. This gives us an equivalence

$$h = (\!|\alpha|\!) \equiv h \circ in^F = \alpha \circ F.h \tag{2.4.1}$$

that completely characterizes catamorphisms. We will refer to this rule in proofs as *catamorphism characterization.*

An immediate consequence of this is a simple but useful rule which we call *catamorphism self*:

$$(\!| in^F |\!) = Id_{\mu F} \tag{2.4.2}$$

and can be proved:

$$\begin{aligned}
& (\!| in^F |\!) = Id_{\mu F} \\
\equiv & \qquad \{ \text{catamorphism characterization} \} \\
& Id_{\mu F} \circ in^F = in^F \circ F.Id_{\mu F} \\
\equiv & \qquad \{ \text{identities, } F \text{ is a functor} \} \\
& in^F = in^F \\
\equiv &
\end{aligned}$$

True

From the uniqueness property we can derive another useful calculational rule called *catamorphism fusion*:

Lemma 2.3: Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor with initial algebra $(\mu F, in^F)$. Suppose $h : A \to B$ is an F-algebra homomorphism between two F-algebras $\alpha : F.A \to A$ and $\beta : F.B \to B$. Then $h \circ \llparenthesis \alpha \rrparenthesis = \llbracket \beta \rrbracket$.

Proof: We know that we have an F-algebra homomorphism $\llparenthesis \alpha \rrparenthesis$ from in^F to α and so we can construct the following *fusion diagram* which is simply the composition of two F-algebra homomorphisms:

$$\begin{tikzcd}
F.\mu F \arrow[r, "F.\llparenthesis \alpha \rrparenthesis"] \arrow[d, "in^F"'] & F.A \arrow[r, "F.h"] \arrow[d, "\alpha"] & F.B \arrow[d, "\beta"] \\
\mu F \arrow[r, dotted, "\llparenthesis \alpha \rrparenthesis"'] & A \arrow[r, "h"'] & B
\end{tikzcd}$$

Thus we see that $h \circ \llparenthesis \alpha \rrparenthesis$ is an F-algebra homomorphism from in^F to β. However, by initiality of in^F we know that there is exactly one F-algebra homomorphism from in^F to β, and that is $\llbracket \beta \rrbracket$. Therefore we can deduce that $h \circ \llparenthesis \alpha \rrparenthesis = \llbracket \beta \rrbracket$.

Written calculationally, this rule is

$$h \circ \alpha = \beta \circ F.h \Rightarrow h \circ \llparenthesis \alpha \rrparenthesis = \llbracket \beta \rrbracket$$

Again, the left-hand side simply states that h is an F-algebra homomorphism.

Let us now look at an example of how these laws can be used in practice, and how they can produce large gains in computational efficiency. We prove a standard result that relates catamorphisms and products:

Lemma 2.4 ("Banana–split" law): Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor with initial algebra $(\mu F, in^F)$, and $\alpha : F.A \to A$ and $\beta : F.B \to B$ two F-algebras. Then

$$\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis = \llparenthesis (\alpha \times \beta) \circ (F.\grave{\pi} \vartriangle F.\acute{\pi}) \rrparenthesis$$

Proof: Using *catamorphism characterization*, we see that

$$\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis = \llparenthesis (\alpha \times \beta) \circ (F.\grave{\pi} \vartriangle F.\acute{\pi}) \rrparenthesis$$

$\equiv$ { catamorphism characterization }

$$(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis) \circ in^F = (\alpha \times \beta) \circ (F.\grave{\pi} \vartriangle F.\acute{\pi}) \circ F.(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis)$$

We can prove the last equality as follows:

$$(\alpha \times \beta) \circ (F.\grave{\pi} \vartriangle F.\acute{\pi}) \circ F.(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis)$$

$=$ { split fusion }

$$(\alpha \times \beta) \circ (F.\grave{\pi} \circ F.(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis) \vartriangle F.\acute{\pi} \circ F.(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis))$$

$=$ { F is a functor }

$$(\alpha \times \beta) \circ (F.(\grave{\pi} \circ (\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis)) \vartriangle F.(\acute{\pi} \circ (\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis)))$$

$=$ { products }

$$(\alpha \times \beta) \circ (F.\llparenthesis \alpha \rrparenthesis \vartriangle F.\llparenthesis \beta \rrparenthesis)$$

$=$ { split fusion }

$$(\alpha \circ F.\llparenthesis \alpha \rrparenthesis) \vartriangle (\beta \circ F.\llparenthesis \beta \rrparenthesis)$$

$=$ { catamorphism diagram }

$$(\llparenthesis \alpha \rrparenthesis \circ in^F) \vartriangle (\llparenthesis \beta \rrparenthesis \circ in^F)$$

$=$ { split fusion }

$$(\llparenthesis \alpha \rrparenthesis \vartriangle \llparenthesis \beta \rrparenthesis) \circ in^F$$

The computational intuition for this result is that if we apply two different folds to the same data-structure to produce a pair of result values, then we can compute the same result values using a single fold, the argument of which is constructed from the arguments to the two original folds. The advantage of performing this transformation is that while $(\!|\alpha|\!) \vartriangle (\!|\beta|\!)$ traverses the data-structure twice — once for each catamorphism — packaging the computations into a single catamorphism means that the data-structure is traversed only once.

Bird and de Moor [BirddM97] give many more laws, and examples of how they can be used to calculate efficient functional programs.

2.5. Existence of initial algebras

In the remainder of this chapter we review some of the main definitions and results from the literature about the *existence* of initial algebras. We will use these definitions and results in chapters 3 and 5. The basic theory is well-explained by Manes and Arbib [ManesA86]. Lehmann and Smyth [LehmannS81] describe the process in the particular case of ω-complete pointed partial orders. Smyth and Plotkin [SmythP82] give more general results in the abstract setting of Wand's **O**-categories [Wand79]. Pierce [Pierce91, section 3.4] gives a good introduction to this material, and Bos and Hemerik [BosH88] present a more detailed treatment. Fokkinga and Meijer [FokkingaM91] show how the results can be applied to program calculation.

We assume the reader has a basic knowledge of domain theory; for example, see [Plotkin83, Schmidt86]. For our purposes we will require *ω-complete partial orders* (ω-c.p.o.s) — partial orders that contain the least upper bounds of all ascending ω-chains. An *ω-continuous* function between two such partial orders is one that is monotonic and preserves the least upper bounds of all ascending ω-chains. A *pointed* partial order is one with a least element, usually denoted $\perp$.

The categorical notions of *(co)limits* and *(co)continuity* are used extensively, and the reader should have some familiarity with them; for example, see [MacLane71, Barr90]. Care should be taken to distinguish between the "categorical" colimits of ω-diagrams, and the "posetal" limits of ω-chains that occur in the hom-sets of order-enriched categories. We will occasionally also mention *adjoint functors*.

2.5.1. Basic existence results

The basic result for the existence of initial algebras is a generalization of the well-known Kleene fixed-point theorem:

> **Theorem 2.1 (Kleene fixed-point theorem):** Let $(D, \sqsubseteq)$ be an ω-complete pointed partial order and $\psi : (D, \sqsubseteq) \to (D, \sqsubseteq)$ an ω-continuous function. Then
>
> $$\bigsqcup \left\{ \psi^i(\bot) \right\}_{i \in \omega}$$
>
> is the least fixed-point of ψ.

Generalizing this from posets to categories gives:

> **Theorem 2.2 (generalized Kleene fixed-point theorem):** If $\mathbb{C}$ is an ω-cocomplete category with an initial object $\mathbb{0}$, and $F : \mathbb{C} \to \mathbb{C}$ is an ω-cocontinuous endofunctor then F has an initial algebra, given by the colimit of the ω-diagram:
>
> $$\begin{tikzcd} \mathbb{0} \arrow[r, "!"] & F.\mathbb{0} \arrow[r, "F.!"] & F.F.\mathbb{0} \arrow[r, "F.F.!"] & \cdots \end{tikzcd}$$

Categories in which this theorem can be applied include two important examples:

Set the category of (small) sets and total functions,

ω-**CPPO** the category of ω-complete pointed partial orders and ω-continuous functions.

Both are ω-cocomplete and have initial objects — in **Set** the initial object is the empty set, and in ω-**CPPO** it is the singleton poset $\{ \bot \}$. One can show in both **Set** and ω-**CPPO** that all *polynomial functors* — those built out of identities, constants, products and sums — are ω-cocontinuous, and therefore have initial algebras. Malcolm [Malcolm90] shows further that type functors are ω-cocontinuous in **Set**. Thus all *regular* types can be shown to exist as initial algebras in **Set**. Fokkinga and Meijer [FokkingaM91] prove a similar result for type functors in ω-**CPPO**, but using the order-enriched techniques described in the next section.

Smyth and Plotkin's "Basic Lemma" [SmythP82] gives weaker conditions for the existence of initial algebras in categories that may not have *all* ω-colimits and for functors that may not preserve *all* ω-colimits.

> **Lemma 2.5 ("Basic lemma"):** Let $\mathbb{C}$ be a category with an initial object $\mathbb{0}$. Let $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor on $\mathbb{C}$. Define $D : \omega \to \mathbb{C}$ to be the diagram
>
> $$\mathbb{0} \xrightarrow{!} F.\mathbb{0} \xrightarrow{F.!} F.F.\mathbb{0} \xrightarrow{F.F.!} \cdots$$
>
> Then if D has a colimiting cone $\mu : D \rhd A$ such that $F\mu : FD \rhd F.A$ is also a colimiting cocone for $FD : \omega \to \mathbb{C}$, then F has an initial algebra.

2.5.2. Results in order-enriched categories

The generalized Kleene fixed-point theorem requires proving "global" conditions — ω-cocompleteness of the category (existence of *all* ω-colimits) and ω-cocontinuity of the functor (preservation of *all* ω-colimits), and these global conditions can be difficult or tedious to prove in general. However, the categories that occur in functional programming often have a partial ordering structure on their hom-sets, and Smyth and Plotkin [SmythP82] developed a body of results that exploited this extra structure to make proving the existence of initial algebras easier. They moved the burden of proving "global" properties to the easier problems of proving "local" conditions on the hom-sets of the category.

Categories with ordered hom-sets are neatly captured in the setting of Wand's **O**-categories [Wand79]:

> **Definition 2.4 (order-enriched category, O-category):** A category $\mathbb{C}$ is *order-enriched*, or an **O**-*category*, if
>
> (i) for every pair of objects A and B in $\mathbb{C}$, the hom-set $hom_{\mathbb{C}}(A, B)$ is an ω-c.p.o.;
>
> (ii) composition in $\mathbb{C}$ is ω-continuous in both arguments.

The obvious example of an **O**-category is the category of ω-c.p.o.'s and ω-continuous functions with the usual approximation ordering on functions. We will also need to talk about **O**-categories whose hom-sets have least or bottom elements:

> **Definition 2.5 (O⊥-category):** A category $\mathbb{C}$ is an **O**⊥-*category* if it is an **O**-category and for each pair of objects A and B in $\mathbb{C}$,
>
> (i) the hom-set $hom_{\mathbb{C}}(A, B)$ has a least element $\perp_{A \to B}$ with respect to the ordering;
>
> (ii) $\perp_{A \to B}$ is a post-zero of composition, that is, for any $f : C \to A$,
>
> $$\perp_{A \to B} \circ f = \perp_{C \to B}$$

Now that our hom-sets have least elements we can talk about strict arrows — arrows that preserve those least elements:

> **Definition 2.6 (strict arrow):** In an **O**⊥-category $\mathbb{C}$, an arrow $f : A \to B$ is *strict* if for all objects C in $\mathbb{C}$,

$$f \circ \bot_{C \to A} = \bot_{C \to B}$$

Strictness plays an important role because the results we use can only prove the existence of initial algebras in the *strict subcategory* of an **O**$\bot$-category:

> **Definition 2.7 (strict subcategory):** For an **O**$\bot$-category $\mathbb{C}$, the *strict subcategory,* $\mathbb{C}_\bot$, has the same objects as $\mathbb{C}$ but only the strict morphisms of $\mathbb{C}$.

The strict subcategory is also an **O**$\bot$-category:

> **Lemma 2.6 (strict subcategory is also an O$\bot$-category):** If $\mathbb{C}$ is an **O**$\bot$-category then so is $\mathbb{C}_\bot$.
>
> **Proof:** Assume that $\mathbb{C}$ is an **O**$\bot$-category.
>
> First we show that $\mathbb{C}_\bot$ is an **O**-category. The hom-sets of $\mathbb{C}_\bot$ inherit a partial ordering from $\mathbb{C}$. We must show that these hom-sets are ω-complete. Let $\alpha = \left\{\alpha_i : A \to B\right\}_{i \in \omega}$ be an ascending ω-chain in the hom-set $hom_{\mathbb{C}_\bot}(A, B)$. Then α is also an ascending ω-chain in $hom_{\mathbb{C}}(A, B)$ and therefore has a least upper bound in $hom_{\mathbb{C}}(A, B)$. However, every α_i is strict and the least upper bound of an ascending ω-chain of strict arrows is itself strict, so $\bigsqcup\alpha$ is an element of $hom_{\mathbb{C}_\bot}(A, B)$. Then $\bigsqcup\alpha$ is clearly an upper bound for α in $hom_{\mathbb{C}_\bot}(A, B)$. We can show it is the *least* upper bound for α in $hom_{\mathbb{C}_\bot}(A, B)$ by considering any other upper bound h of α in $hom_{\mathbb{C}_\bot}(A, B)$— then h is also an upper bound of α in $hom_{\mathbb{C}}(A, B)$ and so $\bigsqcup\alpha \sqsubseteq_{\mathbb{C}} h$. Because the ordering is inherited we then have that $\bigsqcup\alpha \sqsubseteq_{\mathbb{C}_\bot} h$.
>
> Because least upper bounds in $\mathbb{C}_\bot$ are just the corresponding least upper bounds in $\mathbb{C}$, it is easy to show that the ω-continuity of composition in $\mathbb{C}$ carries over to composition in $\mathbb{C}_\bot$. Therefore we have shown that $\mathbb{C}_\bot$ is an **O**-category.

We must further show that $\mathbb{C}_\perp$ is an $\mathbf{O}\perp$-category. The least elements of the hom-sets of $\mathbb{C}$ are strict and therefore appear in the hom-sets of $\mathbb{C}_\perp$, where they are also least because the ordering is inherited. Each least element is the post-zero of composition for its hom-set in $\mathbb{C}_\perp$ because composition is inherited from $\mathbb{C}$. Therefore $\mathbb{C}_\perp$ is an $\mathbf{O}\perp$-category.

The main construction is based on Dana Scott's "inverse limit" construction, which involves taking colimits of ω-diagrams of *projection pairs*:

Definition 2.8 (projection pair, projection or retraction, embedding): Let $\mathbb{C}$ be an $\mathbf{O}$-category and let $f : A \to B$ and $g : B \to A$ be arrows such that

(i) $g \circ f = Id_A$

(ii) $f \circ g \sqsubseteq Id_B$

Then we say that (f, g) is a *projection pair* from A to B, that g is a *projection* or *retraction* and that f is an *embedding*.

Projection pairs can be composed, and the identity pairs, (Id_A, Id_A), are the unit of composition. In this way we can consider *categories of projection pairs*:

Definition 2.9 (category of projection pairs): Given an $\mathbf{O}$-category $\mathbb{C}$, we can construct a *category of projection pairs*, $\mathbb{C}_{PR}$, with the same objects as $\mathbb{C}$ and projection pairs as arrows. Composition is given by

$$(f', g') \circ (f, g) = (f' \circ f, g \circ g')$$

Identity arrows are the identity pairs, (Id_A, Id_A), for each A in $\mathbb{C}$.

One half of a projection pair *uniquely determines* the other [Wand79, proposition 2.7], so, given an embedding $f : A \to B$, we write $f^R : B \to A$ to mean the unique corresponding retraction. Conversely, given a retraction $g : B \to A$, we write $g^L : A \to B$

for the corresponding embedding. Because of this, it is sufficient to consider only a category of embeddings:

Definition 2.10 (category of embeddings): Given an **O**-category $\mathbb{C}$, we can construct a *category of embeddings*, $\mathbb{C}_E$, with the same objects as $\mathbb{C}$ but only embeddings as arrows. Composition and identities are the same as in $\mathbb{C}$.

It is straightforward to define functors to pass between $\mathbb{C}_E$ and $\mathbb{C}_{PR}$:

$$S : \mathbb{C}_E \longrightarrow \mathbb{C}_{PR} \qquad\qquad T : \mathbb{C}_{PR} \longrightarrow \mathbb{C}_E$$

$$\begin{tikzcd}
A \arrow[d, "f"'] \arrow[dr, phantom, "\mapsto"] & A \arrow[d, "{(f,f^R)}"] & & A \arrow[d, "{(f,f^R)}"'] \arrow[dr, phantom, "\mapsto"] & A \arrow[d, "f"] \\
B & B & & B & B
\end{tikzcd}$$

and they are clearly inverses of each other; thus we have an *isomorphism of categories* [MacLane71, section IV.4]. This means that S and T are both left and right adjoints of each other, and therefore preserve limits and colimits, so we can choose to work in either category as convenient. We can also define a *category of retractions* $\mathbb{C}_R$, again isomorphic to $\mathbb{C}_E$ and therefore also $\mathbb{C}_{PR}$.

We will need our categories of embeddings to be ω-cocomplete and the following lemma gives some sufficient conditions on $\mathbb{C}$ for this:

Lemma 2.7: Let $\mathbb{C}$ be an **O**-category, then $\mathbb{C}_E$ is ω-cocomplete if either of the following is true:

(i) $\mathbb{C}$ is ω-cocomplete;

(ii) $\mathbb{C}$ is ω^{op}-complete.

Proof: Let $D : \omega \to \mathbb{C}_E$ be any ω-diagram in $\mathbb{C}_E$. We prove the two cases above separately:

(i) If $\mathbb{C}$ is ω-cocomplete then D has a colimit in $\mathbb{C}$, and then by Smyth and Plotkin's theorem 2 [SmythP82] we know that D has a colimit in $\mathbb{C}_E$, so $\mathbb{C}_E$ is ω-cocomplete.

(ii) From D we can construct an ω^{op}-diagram, $D_R : \omega^{op} \to \mathbb{C}_R$ containing the corresponding retractions to the embeddings making up D. If $\mathbb{C}$ is ω^{op}-complete then D_R has a limit in $\mathbb{C}$ and again by Smyth and Plotkin's theorem 2 we know that D has a colimit in $\mathbb{C}_E$, so $\mathbb{C}_E$ is ω-cocomplete.

We also know that embeddings and retractions in an **O**$\perp$-category are always strict:

Lemma 2.8: Let $\mathbb{C}$ be an **O**$\perp$-category, and (f^L, f^R) a projection pair from A to B in $\mathbb{C}$. Then both f^L and f^R are strict.

Proof: We prove, for any C in $\mathbb{C}$, that f^L is strict:

True

$\equiv$ { $\perp_{C\to A}$ is the least element of $hom_{\mathbb{C}}(C, A)$ }

$\perp_{C\to A} \sqsubseteq f^R \circ \perp_{C\to B}$

$\Rightarrow$ { $(f^L\circ)$ is monotonic }

$f^L \circ \perp_{C\to A} \sqsubseteq f^L \circ f^R \circ \perp_{C\to B}$

$\Rightarrow$ { $f^L \circ f^R \sqsubseteq Id_B$ and $(\circ\perp_{C\to B})$ is monotonic }

$f^L \circ \perp_{C\to A} \sqsubseteq Id_B \circ \perp_{C\to B}$

$\equiv$ { identity }

$f^L \circ \perp_{C\to A} \sqsubseteq \perp_{C\to B}$

$\Rightarrow$ { $\perp_{C\to B}$ is the least element of $hom_{\mathbb{C}}(C, B)$ }

$f^L \circ \perp_{C\to A} = \perp_{C\to B}$

The proof for f^R is similar.

This gives us the immediate corollary:

Corollary 2.1: If $\mathbb{C}$ is an $\mathbf{O}\perp$-category then

(i) $\left(\mathbb{C}_\perp\right)_{PR} = \mathbb{C}_{PR}$

(ii) $\left(\mathbb{C}_\perp\right)_E = \mathbb{C}_E$

(iii) $\mathbb{C}_E \subseteq \mathbb{C}_\perp$

So we have the following relationships between our derived categories:

$$\mathbb{C}_{PR} \cong \mathbb{C}_E \subseteq \mathbb{C}_\perp \subseteq \mathbb{C}$$

We want to transfer the burden of proving that a functor is “globally” ω-cocontinuous to proving that it is “locally” ω-continuous on the hom-sets of the category:

Definition 2.11 (locally ω-continuous functor): If $\mathbb{C}$ and $\mathbb{D}$ are **O**-categories then a functor $F : \mathbb{C} \to \mathbb{D}$ is *locally ω-continuous* if for every pair of objects A and B in $\mathbb{C}$, the restriction of F’s action on arrows to the hom-set $hom_{\mathbb{C}}(A, B)$ is ω-continuous.

This “local” ω-continuity is easier to establish in general. A related notion to that of local ω-continuity is *local monotonicity*:

Definition 2.12 (locally monotonic functor): Let $F : \mathbb{C} \to \mathbb{D}$ be a functor between two **O**-categories. We say F is *locally monotonic* if for any two arrows f and g in a particular hom-set of $\mathbb{C}$,

$$f \sqsubseteq_{\mathbb{C}} g \Rightarrow F.f \sqsubseteq_{\mathbb{D}} F.g$$

Clearly, locally ω-continuous functors are also locally monotonic.

Many of the standard constructions on ω-complete pointed partial orders, such as Cartesian product, smash product, coalesced sums and separated sums, are locally ω-continuous. Furthermore, they *preserve* local ω-continuity when lifted:

Lemma 2.9: If $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ is a locally ω-continuous bifunctor and $F, G : \mathbb{B} \to \mathbb{C}$ are two locally ω-continuous functors then $F \dot{\otimes} G : \mathbb{B} \to \mathbb{C}$ is also locally ω-continuous.

Proof: We first of all establish local monotonicity. Let $f, g : A \to B$ be two arrows in $hom_{\mathbb{B}}(A, B)$ such that $f \sqsubseteq_{\mathbb{B}} g$, then

$$
\begin{array}{ll}
 & (F \dot{\otimes} G).f \\
= & \quad \{ \text{definition of } \dot{\otimes} \} \\
 & F.f \otimes G.f \\
\sqsubseteq_{\mathbb{C}} & \quad \{ F, G \text{ and } \otimes \text{ are locally monotonic} \} \\
 & F.g \otimes G.g \\
= & \quad \{ \text{definition of } \dot{\otimes} \} \\
 & (F \dot{\otimes} G).g
\end{array}
$$

so $F \dot{\otimes} G$ is locally monotonic.

Let $\left\{f_i : A \to B\right\}_{i \in \omega}$ be an ascending ω-chain in $hom_{\mathbb{B}}(A, B)$. Then $\left\{(F \dot{\otimes} G).f_i\right\}_{i \in \omega}$ is also ascending and therefore has a least upper bound. Then we calculate:

$$
\begin{array}{ll}
 & (F \dot{\otimes} G).\sqcup\left\{f_i\right\}_{i \in \omega} \\
= & \quad \{ \text{definition of } \dot{\otimes} \} \\
 & F.\sqcup\left\{f_i\right\}_{i \in \omega} \otimes G.\sqcup\left\{f_i\right\}_{i \in \omega}
\end{array}
$$

$$
\begin{array}{ll}
= & \{\, F \text{ and } G \text{ are locally } \omega\text{-continuous} \,\} \\
& \bigsqcup\left\{F.f_i\right\}_{i\in\omega} \otimes \bigsqcup\left\{G.f_i\right\}_{i\in\omega} \\
= & \{\, \otimes \text{ is locally } \omega\text{-continuous} \,\} \\
& \bigsqcup\left\{F.f_i \otimes G.f_i\right\}_{i\in\omega} \\
= & \{\, \text{definition of } \dot{\otimes} \,\} \\
& \bigsqcup\left\{(F \mathbin{\dot{\otimes}} G).f_i\right\}_{i\in\omega}
\end{array}
$$

and so $F \mathbin{\dot{\otimes}} G$ is locally ω-continuous.

A nice property of locally monotonic functors is that they preserve projection pairs, and therefore also embeddings:

Lemma 2.10 (locally monotonic functors preserve projection pairs): Let $F : \mathbb{C} \to \mathbb{D}$ be a locally monotonic functor between two **O**-categories, and (f, f^R) a projection pair from A to B in $\mathbb{C}$. Then $(F.f, F.f^R)$ is a projection pair from $F.A$ to $F.B$ in $\mathbb{D}$.

Proof: The properties of a projection pair are established by two simple calculations:

(i)

$$
\begin{array}{ll}
& F.f^R \circ F.f \\
= & \{\, F \text{ is a functor} \,\} \\
& F.(f^R \circ f) \\
= & \{\, f^R \text{ is the retraction of } f \,\} \\
& F.Id_A \\
= & \{\, F \text{ is a functor} \,\} \\
& Id_{F.A}
\end{array}
$$

(ii)

$$
\begin{array}{ll}
 & F.f \circ F.f^R \\
= & \{ F \text{ is a functor} \} \\
 & F.(f \circ f^R) \\
\sqsubseteq_{\mathbb{D}} & \{ f \circ f^R \sqsubseteq_{\mathbb{C}} Id_B \text{ and } F \text{ is locally monotonic} \} \\
 & F.Id_B \\
= & \{ F \text{ is a functor} \} \\
 & Id_{F.B}
\end{array}
$$

So, given any locally monotonic functor $F : \mathbb{C} \to \mathbb{D}$, we can derive a new functor that operates on projection pairs:

$$\begin{tikzcd}
F_{PR} : \mathbb{C}_{PR} \arrow[r, ""] & \mathbb{D}_{PR} \\
A \arrow[d, "(h^L, h^R)"'] \arrow[r, phantom, "\mapsto"] & F.A \arrow[d, "(F.h^L, F.h^R)"] \\
B & F.B
\end{tikzcd}$$

or even just the restriction of F to embeddings:

$$\begin{tikzcd}
F_E : \mathbb{C}_E \arrow[r] & \mathbb{D}_E \\
A \arrow[d, "f"'] \arrow[r, phantom, "\mapsto"] & F.A \arrow[d, "F.f"] \\
B & F.B
\end{tikzcd}$$

because we know that locally monotonic functors preserve projection pairs and embeddings.

Freyd [Freyd90] shows that any locally ω-continuous functor $F : \mathbb{C} \to \mathbb{D}$ between two **O**$\perp$-categories preserves strict arrows provided that $\mathbb{C}$ has a terminal object. Freyd uses a variation of the notion of **O**$\perp$-category, but we reformulate his results to match the definitions in this chapter. First note the following two facts:

Lemma 2.11: If $\mathbb{C}$ is an **O**$\perp$-category with a terminal object $\mathbb{1}$ then for any

A and B in $\mathbb{C}$, the least element of $hom_{\mathbb{C}}(A, B)$ can be factorized:

$$\perp_{A \to B} = A \xrightarrow{!_A} \mathbb{1} \xrightarrow{\perp_{\mathbb{1} \to B}} B$$

Proof: Because $\perp$ is the post-zero of composition.

Lemma 2.12: Let $\mathbb{C}$ be an **O**$\perp$-category and $f, g : A \to B$ two arrows in $\mathbb{C}$. If $f \sqsubseteq g$ and g is strict then f must also be strict.

Proof: For any D in $\mathbb{C}$:

$$
\begin{array}{ll}
 & f \circ \perp_{D \to A} = \perp_{D \to B} \\
\equiv & \quad \{ \perp_{D \to B} \text{ is the least element in } hom_{\mathbb{C}}(D, B) \} \\
 & f \circ \perp_{D \to A} \sqsubseteq \perp_{D \to B} \\
\equiv & \quad \{ g \text{ is strict} \} \\
 & f \circ \perp_{D \to A} \sqsubseteq g \circ \perp_{D \to A} \\
\Leftarrow & \quad \{ (\circ \perp_{D \to A}) \text{ is monotonic} \} \\
 & f \sqsubseteq g
\end{array}
$$

So f is strict.

We also need the "cancellation lemma":

Lemma 2.13 ("cancellation lemma", [Freyd90]): If $\mathbb{C}$ is an **O**$\perp$-category and the composite arrow

$$A \xrightarrow{f} B \xrightarrow{g} C$$

is strict in $\mathbb{C}$ then g must be strict.

Proof: Assume $g \circ f$ is strict, so for any D in $\mathbb{C}$,

$$g \circ f \circ \perp_{D \to A} = \perp_{D \to C} \tag{2.5.1}$$

Then we calculate

$$
\begin{array}{ll}
 & g \circ \bot_{D \to B} = \bot_{D \to C} \\
\equiv & \qquad \{\ \bot_{D \to C} \text{ is the least element of } hom_{\mathbb{C}}(D, C)\ \} \\
 & g \circ \bot_{D \to B} \sqsubseteq \bot_{D \to C} \\
\equiv & \qquad \{\ \text{equation 2.5.1}\ \} \\
 & g \circ \bot_{D \to B} \sqsubseteq g \circ f \circ \bot_{D \to A} \\
\Leftarrow & \qquad \{\ (g\, \circ\,) \text{ is monotonic}\ \} \\
 & \bot_{D \to B} \sqsubseteq f \circ \bot_{D \to A} \\
\equiv & \qquad \{\ \bot_{D \to B} \text{ is the least element of } hom_{\mathbb{C}}(D, B)\ \} \\
 & \text{True}
\end{array}
$$

So g is strict.

Now we can prove Freyd's lemma:

Lemma 2.14 ([Freyd90]): Let $\mathbb{C}$ and $\mathbb{D}$ be $\mathbf{O}\bot$-categories and let $\mathbb{C}$ have a terminal object $\mathbb{1}$. If $T : \mathbb{C} \to \mathbb{D}$ is a locally monotonic functor then it preserves strict arrows.

Proof: Consider any B in $\mathbb{C}$, then

$$
\begin{array}{ll}
 & \text{True} \\
\equiv & \qquad \{\ \bot_{B \to B} \text{ is the least element of } hom_{\mathbb{C}}(B, B)\ \} \\
 & \bot_{B \to B} \sqsubseteq_{\mathbb{C}} Id_B \\
\Rightarrow & \qquad \{\ T \text{ is locally monotonic}\ \} \\
 & T.\bot_{B \to B} \sqsubseteq_{\mathbb{D}} T.Id_B \\
\equiv & \qquad \{\ T \text{ is a functor}\ \} \\
 & T.\bot_{B \to B} \sqsubseteq_{\mathbb{D}} Id_{T.B} \\
\Rightarrow & \qquad \{\ \text{identities are strict, lemma 2.12}\ \} \\
 & T.\bot_{B \to B} \text{ is strict}
\end{array}
$$

But $\bot_{B \to B}$ can be factorized using lemma 2.11:

$$\perp_{B\to B} = B \xrightarrow{!_B} \mathbb{1} \xrightarrow{\perp_{\mathbb{1}\to B}} B$$

and applying T gives us:

$$T.\perp_{B\to B} = T.B \xrightarrow{T.!_B} T.\mathbb{1} \xrightarrow{T.\perp_{\mathbb{1}\to B}} T.B$$

which we have proved to be strict. Then the cancellation lemma (lemma 2.13) tells us that $T.\perp_{\mathbb{1}\to B}$ must also be strict.

Now consider any strict arrow $f : A \to B$ in $\mathbb{C}$. We calculate:

f is strict

$\Rightarrow$ { definition of strict }

$f \circ \perp_{\mathbb{1}\to A} = \perp_{\mathbb{1}\to B}$

$\Rightarrow$ { T is a functor }

$T.f \circ T.\perp_{\mathbb{1}\to A} = T.\perp_{\mathbb{1}\to B}$

$\Rightarrow$ { $T.\perp_{\mathbb{1}\to B}$ is strict }

$T.f \circ T.\perp_{\mathbb{1}\to A}$ is strict

$\Rightarrow$ { cancellation lemma (lemma 2.13) }

$T.f$ is strict

and so T preserves strict arrows.

Corollary 2.2: If $T : \mathbb{C} \to \mathbb{D}$ is a locally ω-continuous functor between two **O**$\perp$-categories and $\mathbb{C}$ has a terminal object, then T can safely be restricted to operate on the strict subcategories:

$$T_\perp : \mathbb{C}_\perp \to \mathbb{D}_\perp$$

$T_\perp$ is also locally ω-continuous.

There are some technical conditions that we need on our **O**-categories in order to make the connection between local ω-continuity of functors and "global" ω-cocontinuity of those functors when restricted to the subcategory embeddings. Smyth and Plotkin [SmythP82] define the notion of **O**-colimit:

> **Definition 2.13 (O-colimits):** Let $\mathbb{C}$ be an **O**-category, $D : \omega \to \mathbb{C}_E$ an ω-diagram in the category of embeddings $\mathbb{C}_E$, and $\mu : D \rhd A$ a cocone under D. Let X be the set of arrows
>
> $$X \overset{def}{=} \left\{ A \xrightarrow{(\mu_i)^R} D.i \xrightarrow{\mu_i} A \right\}_{i \in \omega}$$
>
> Then μ is an **O**-*colimit of* D if
>
> (i) X is an ascending ω-chain with respect to the ordering on $hom_{\mathbb{C}}(A, A)$;
>
> (ii) $\bigsqcup X = Id_A$.

However, Bos and Hemerik [BosH88] show that condition (i) is, in fact, redundant. They give the lemma:

> **Lemma 2.15 ([BosH88, lemma 3.9]):** Let $\mathbb{C}$ be an **O**-category, and $D : \omega \to \mathbb{C}_{PR}$ an ω-diagram in $\mathbb{C}_{PR}$. Then, for any cocones $\alpha : D \rhd A$ and $\beta : D \rhd B$ under D, the ω-indexed set,
>
> $$\left\{ B \xrightarrow{(\beta_i)^R} D.i \xrightarrow{(\alpha_i)^L} A \right\}_{i \in \omega}$$
>
> is an ascending ω-chain in $\hom_{\mathbb{C}}(B, A)$.

which, if we instantiate with both α and β being the cocone μ from the definition of **O**-colimit above, shows that condition (i) is always satisfied. Thus we have a simpler characterization of **O**-colimit:

Lemma 2.16: Let $\mathbb{C}$ be an **O**-category and $D : \omega \to \mathbb{C}_E$ an ω-diagram in $\mathbb{C}_E$. Then a cocone $\mu : D \rhd A$ under D is an **O**-colimit of D if and only if

$$\bigsqcup \left\{ A \xrightarrow{(\mu_i)_R} D.i \xrightarrow{\mu_i} A \right\}_{i \in \omega} = Id_A$$

The required condition on the **O**-category is then that ω-colimits of embeddings coincide with **O**-colimits:

Definition 2.14 (locally determined ω-colimits of embeddings): An **O**-category is said to have *locally determined ω-colimits of embeddings* if a cocone $\mu : D \rhd A$ is a colimit of an ω-diagram $D : \omega \to \mathbb{C}_E$ if and only if μ is an **O**-colimit of D.

A sufficient condition for an **O**-category to have localized ω-colimits of embeddings is for the category to be ω^{op}-complete:

Lemma 2.17 ([SmythP82]): If $\mathbb{C}$ is an **O**-category that has limits of all ω^{op}-diagrams then $\mathbb{C}$ has locally determined ω-colimits of embeddings.

Bos and Hemerik [BosH88] introduce the above technical condition in a slightly different form — they define the notion of a *localized* category:

Definition 2.15 (localized category [BosH88, definition 3.11]): An **O**-category $\mathbb{C}$ is *localized* if for any ω-diagram $D : \omega \to \mathbb{C}_{PR}$ with colimiting cocone $\alpha : D \rhd A$ in $\mathbb{C}_{PR}$, then there is an object B in $\mathbb{C}$ and a projection pair $\left(h^L, h^R\right)$ from B to A such that

$$h^L \circ h^R = \bigsqcup \left\{ A \xrightarrow{(\alpha_i)^R} D.i \xrightarrow{(\alpha_i)^L} A \right\}_{i \in \omega}$$

but this is equivalent to Smyth and Plotkin's notion of a category having locally determined ω-colimits of embeddings:

Lemma 2.18: Let $\mathbb{C}$ be an **O**-category. Then $\mathbb{C}$ is localized if and only if it has locally determined ω-colimits of embeddings.

Proof:

Case ($\Rightarrow$):

Let $\mathbb{C}$ be localized. Let $D : \omega \to \mathbb{C}_E$ be an ω-diagram and $\alpha : D \rhd A$ a cocone under D. We must show that α is a colimit of D if and only if α is an **O**-colimit of D. We reason:

α is a colimit of D

$\equiv$ { [BosH88, "initiality theorem", theorem 3.12] }

$\bigsqcup\left\{\alpha_i \circ (\alpha_i)^R\right\}_{i\in\omega} = Id_A$

$\equiv$ { lemma 2.16 }

α is an **O**-colimit of D

Case ($\Leftarrow$):

Suppose $\mathbb{C}$ has locally determined ω-colimits of embeddings. Let $D : \omega \to \mathbb{C}_E$ be an ω-diagram in $\mathbb{C}_E$ and $\mu : D \rhd A$ a colimiting cocone under D. Then we must find an object B in $\mathbb{C}$ and a projection pair (f, f^R) from B to A such that

$$\bigsqcup\left\{\mu_i \circ (\mu_i)^R\right\}_{i\in\omega} = f \circ f^R$$

Because $\mathbb{C}$ has locally determined ω-colimits of embeddings, μ is an **O**-colimit of D and therefore:

$$\bigsqcup\left\{\mu_i \circ (\mu_i)^R\right\}_{i\in\omega} = Id_A$$

The obvious choice for our projection pair is then (Id_A, Id_A), and we are finished.

Examples of localized categories include not only ω-**CPPO** [BosH88, theorem 3.22] but also the following (see [BosH88, theorem 3.26]):

D-CPPO the category of *directed-complete pointed partial orders* and *directed-continuous maps*;

CL the category of *complete lattices* and *directed-complete maps*.

The product of two localized categories is a localized category [BosH88, proposition 3.21] and the opposite category of a localized category is a localized category [BosH88, corollary 3.20]. Furthermore, strict subcategories of localized **O**$\perp$-categories are also localized:

Lemma 2.19: If $\mathbb{C}$ is a localized **O**$\perp$-category then so is $\mathbb{C}_\perp$.

Proof: Let $\mathbb{C}$ be a localized **O**$\perp$-category. Lemma 2.6 tells us that $\mathbb{C}_\perp$ is also an **O**$\perp$-category, so we only need to show that it is localized. The conditions required for $\mathbb{C}_\perp$ to be localized are actually conditions on $\left(\mathbb{C}_\perp\right)_{PR}$. However, corollary 2.1 tells us that $\left(\mathbb{C}_\perp\right)_{PR}$ is equal to $\mathbb{C}_{PR}$, and the conditions are satisfied by $\mathbb{C}_{PR}$ because $\mathbb{C}$ is localized. Therefore $\mathbb{C}_\perp$ is localized.

We need to calculate colimits of ω-diagrams in $\mathbb{C}_{PR}$ or equivalently in $\mathbb{C}_E$. To do this, it will be helpful if $\mathbb{C}_E$ has an initial object, and Smyth and Plotkin give a sufficient condition on $\mathbb{C}$ to ensure this:

Theorem 2.3 ([SmythP82, theorem 1]): If $\mathbb{C}$ is an **O**$\perp$-category with a terminal object $\mathbb{1}$, then $\mathbb{1}$ is initial in $\mathbb{C}_E$.

We will additionally make use of the following fact about terminal objects:

Lemma 2.20: If $\mathbb{C}$ is an **O**$\perp$-category with a terminal object $\mathbb{1}$, then $\mathbb{1}$ is also terminal in $\mathbb{C}_{\perp}$.

Proof: It is sufficient to show that for any object A in $\mathbb{C}$ that the unique terminal arrow $!_A : A \to \mathbb{1}$ is strict, and so appears in $\mathbb{C}_{\perp}$. For any B in $\mathbb{C}$,

$$\begin{array}{ll} & !_A \circ \perp_{B\to A} \\ = & \qquad \{\text{ terminal arrow }\} \\ & !_B \\ = & \qquad \{\text{ hom-set has only one element so it must be least }\} \\ & \perp_{B\to\mathbb{1}} \end{array}$$

So the terminal arrows are strict.

We are now in a position to state the main results we need. These are neatly summarized in the first three parts of Fokkinga and Meijer's [FokkingaM91] main theorem:

Theorem 2.4 ([FokkingaM91, main theorem]): Let $\mathbb{C}$ be a localized **O**$\perp$-category such that $\mathbb{C}_{PR}$ is ω-cocomplete and has an initial object, and let $F : \mathbb{C}_{\perp} \to \mathbb{C}_{\perp}$ be a locally ω-continuous functor. Then,

(i) there exists an object μF in $\mathbb{C}$ and strict morphisms $in^F : F.\mu F \to \mu F$ and $out^F : \mu F \to F.\mu F$ which are each others inverses; furthermore, $Id_{\mu F}$ is given by the least fixed-point of the mapping

$$h \mapsto in^F \circ F.h \circ out^F$$

(ii) μF is unique up to (a unique) isomorphism, and, given μF, in^F and out^F are unique too;

(iii) $(\mu F, in^F)$ is initial in $Alg(F)$, and for any (A, ϕ) in $Alg(F)$, the unique strict algebra homomorphism $(\!|\phi|\!) : (\mu F, in^F) \to (A, \phi)$ is the least solution for h of $h \circ in^F = \phi \circ F.h$ (in $\mathbb{C}$ as well as in $\mathbb{C}_\perp$);

This theorem can be used to show the existence of initial algebras for locally ω-continuous functors. The categories in which this theorem is usually applied are ω-**CPPO** and ω-**CPPO**$_\perp$, and the standard constructions used to build data-types in these categories, such as *products, smash products, separated sums* and *coalesced sums*, and *lifting* all give rise to locally ω-continuous functors [Wand79, LehmannS81, SmythP82]. As a result, all *polynomial* functors have initial algebras. Furthermore, for parameterized types, Fokkinga and Meijer [FokkingaM91] show that type functors are also locally ω-continuous, and so all *regular* functors have initial algebras.

We will also use another theorem that derives from Plotkin's "Structural Induction Theorem" [Plotkin83]:

Theorem 2.5: Under the same conditions as theorem 2.4 above, if $\alpha : F.A \to A$ is an F-algebra, then the following are equivalent:

(i) α is an initial F-algebra;

(ii) α is an isomorphism and the least fixed-point of the mapping

$$f \mapsto \alpha \circ F.f \circ \alpha^{-1}$$

is the identity on A.

2.6. Algebraic data-types in functional programming

In this section we will see how the categorical theory we have described relates to the mechanisms for recursive type definition in modern functional programming languages.

Haskell and ML provide the programmer with *algebraic data-types*. Data-types are defined by giving the names and signatures of the constructors. For instance, in Haskell, integer lists can be defined as

```
data  List  =  NilL
            |  ConsL  Int  List
```

This would introduce a new type List and two constructor functions:

$$\mathsf{NilL} :: \mathsf{List}$$

$$\mathsf{ConsL} :: \mathsf{Int} \rightarrow \mathsf{List} \rightarrow \mathsf{List}$$

Functions are curried by default in Haskell, but if we uncurry ConsL and treat NilL as a constant function then we can see that the types correspond to the categorical constructors

$$\mathsf{NilL}_{Int} : \mathbb{1} \rightarrow List_{Int}$$

$$\mathsf{ConsL}_{Int} : Int \times List_{Int} \rightarrow List_{Int}$$

which form an algebra for the functor $\underline{\mathbb{1}} \dot{+} \underline{Int} \dot{\times} Id$. Furthermore, pattern-matching provides an inverse to the constructors in that given any value of the List type we can use pattern-matching to determine which constructor function was used to construct the value and also the arguments to that constructor function.

The catamorphism pattern of recursion can be coded directly into Haskell. The catamorphism diagram gives us a point-free definition of $(\![\, \lambda \triangledown \rho \,]\!)$:

$$(\![\, \lambda \triangledown \rho \,]\!) \circ \mathsf{NilL}_{Int} = \lambda$$

$$(\![\, \lambda \triangledown \rho \,]\!) \circ \mathsf{ConsL}_{Int} = \rho \circ Id \times (\![\, \lambda \triangledown \rho \,]\!)$$

Translating this into a point-wise definition and parameterizing the catamorphism arguments gives us a combinator

```
foldList lam rho NilL = lam
foldList lam rho (ConsL x xs) = rho x (foldList lam rho xs)
```

The type system can infer the type of the foldList combinator as

```
foldList :: a → (Int → a → a) → List → a
```

Furthermore, we can show that folding with the list constructors:

```
(foldList NilL ConsL) :: List → List
```

is the identity on lists, because it simply replaces the list constructors by themselves. For finite lists, this is easy enough to see, and can be proved by induction on the length of the list. For infinite lists we can use the fact that an infinite list is the least upper bound of an ascending ω-chain of finite approximations, and (foldList NilL ConsL) is the identity on all those finite approximations, so it must also be the identity on the least upper bound, because it is ω-continuous.

If we take F to be the functor $\underline{\mathbb{1}} \dot{+} \underline{Int} \dot{\times} Id$, α to be $\mathsf{NilN} \triangledown \mathsf{ConsN}$ and α^{-1} to correspond to pattern-matching on the list constructors, then (foldList NilL ConsL) is the least fixed-point of

$$f \mapsto \alpha \circ F.f \circ \alpha^{-1}$$

which we have shown to be the identity on lists, so condition (ii) of theorem 2.5 is satisfied and we can conclude that the constructors constitute an initial F-algebra. In other words, (foldList lam rho) is the catamorphism from $\mathsf{NilN} \triangledown \mathsf{ConsN}$ to some other F-algebra $\mathsf{lam} \triangledown \mathsf{rho}$ (provided that $\mathsf{lam} \triangledown \mathsf{rho}$ is strict). The same can be shown for other regular types.

Haskell and ML also allow the definition of *parameterized* data-types by the introduction of type variables. Parameterized lists could be defined

```
data List a = NilL
             | ConsL a (List a)
```

Instead of introducing a new type, this defines a *type constructor* List, and two *polymorphic* constructors

```
NilL :: List a
ConsL :: a → List a → List a
```

Categorically, the type constructor List corresponds to the type functor for the list bifunctor in section 2.3. The instances of the polymorphic constructors correspond to the initial algebras that arise when one argument of the bifunctor is fixed.

The operation of the map combinator for a regular parameterized data-type is simple to translate into a Haskell function. Although we saw that maps are defined using catamorphisms in the categorical theory, in practice, maps are usually simplified and given directly as recursive functions. For example, the map combinator for lists would be

```
mapList :: (a → b) → List a → List b
mapList f NilL = NilL
mapList f (ConsL x xs) = ConsL (f x) (mapList f xs)
```

We can see that the recursive structure is copied — NilL is replaced by NilL and ConsL by ConsL — and the argument f is applied to each value of the parameter type. Maps can be derived easily for other regular data-types.

However, maps and folds are only derivable for *regular* types — those that can be formed as initial algebras of functors built out of sums, products, constants, identities, and type functors. This covers a wide class of useful data-types but excludes significant types such as

- types that use the function space type constructor contravariantly;

- non-uniform data-types.

Attempts have been made to define fold operators for types involving function spaces [MeijerH95, FegarasS95]. In this thesis we address the question of folds for non-uniform data-types.

2.7. Chapter summary

In this chapter we have described the traditional categorical theory of data-types as initial algebras. The types of a data-type's constructors determine the signature or *base functor*, and the constructors themselves form an *initial algebra* for the base functor. Initiality ensures a unique *catamorphism* to any other algebra, which captures exactly the *fold* pattern of recursion. Initiality also provides calculational laws for catamorphisms such as *fusion*.

Parameterized data-types are treated by fixing the parameter for every possible parameter value and forming the initial algebras for the resulting non-parameterized types. Taken together, these initial algebras give rise to a *type functor*, whose action on arrows captures the *map* pattern of recursion for a particular parameterized type.

In order to use catamorphisms and their calculational laws, it is necessary to first show that an initial algebra exists. There are at least two ways to try to establish the existence of initial algebras for particular base functors in particular categories — the generalized Kleene fixed-point theorem and the order-enriched results of Smyth and Plotkin [SmythP82].

Chapter 3. Folding in functor categories

A value of a recursive data-type may be built using an arbitrary number of constructors. In the case of *uniform* parameterized data-types, because the recursive calls are all made at the same parameter instance, all the constructors will also be at the same instance. Therefore, to fold over a uniform parameterized type, it suffices to provide functions to replace a single instance of the constructors, and these are given as arguments to the fold. The important difference with *non-uniform* data-types is that the constructors used to build a value need not all be at the same instance. In fact, all the constructors will usually occur at *different* instances. The problem is then to produce replacements for *all* the different constructor instances, and make them available to the fold in some manner.

One possible solution, proposed independently by Jones [Jones98] and Bird and Meertens [BirdM98], is to exploit the capabilities for *polymorphism* already available in the type system. The solution is not to give single functions as arguments to the fold, but *polymorphic functions* — uniform *families* of functions indexed by type. The constructor functions for the data-type are themselves polymorphic families of functions, and a fold combinator could then replace instances of the polymorphic constructors by the corresponding instances of the polymorphic replacement functions.

Taking the approach of folding with *polymorphic* functions relates naturally to the initial algebra semantics of data-types we described in the last chapter, except that everything is lifted to the level of *functor categories*.

3.1. Natural transformations and functor categories

In a first-order language, (strict) parametric polymorphic functions can be viewed as *natural transformations* between functors corresponding to type constructors [Wadler89]. If we are to consider folds as catamorphisms where the algebras are polymorphic functions then we need a category whose arrows are natural transformations. Fortunately such categories have been well-studied and are called *functor categories*:

> **Definition 3.1 (functor category):** For a *small* category $\mathbb{C}$ and an arbitrary category $\mathbb{D}$, the *functor category* $\mathbb{D}^{\mathbb{C}}$ is defined as follows:
>
> - objects in $\mathbb{D}^{\mathbb{C}}$ are the functors from $\mathbb{C}$ to $\mathbb{D}$;
> - for two objects F and G in $\mathbb{D}^{\mathbb{C}}$, the arrows between them are the natural transformations $F \xrightarrow{\cdot} G$;
> - composition of arrows is the vertical or componentwise composition of natural transformations, that is, for $\alpha : F \xrightarrow{\cdot} G$ and $\beta : G \xrightarrow{\cdot} H$, the components of the composite are given by:
>
> $$(\beta \circ \alpha)_A \overset{def}{=} \beta_A \circ \alpha_A$$
>
>   for each A in $\mathbb{C}$;
> - identity arrows are the identity natural transformations (those in which all the components are identity arrows in $\mathbb{D}$).

A *small* category is one whose class of arrows (and therefore also objects) is a set. The restriction that $\mathbb{C}$ be small is required to avoid problems in the set-theoretic foundations of category theory. It is interesting to note that Mac Lane [MacLane88] sees this as a problem that logicians have failed to address:

> "There is a need for an effective foundation for the convenient handling of such large constructions as the category of all categories and functor categories of large categories."

3.1.1. Evaluation functors

Each functor category $\mathbb{D}^{\mathbb{C}}$ has an *evaluation functor* that applies a functor to an argument:

> **Definition 3.2 (evaluation functor):** For a functor category $\mathbb{D}^{\mathbb{C}}$, the *evaluation functor* is defined:

$$\begin{tikzcd}
@ : \mathbb{D}^{\mathbb{C}} \times \mathbb{C} \arrow[r] & \mathbb{D} \\
(F, A) \arrow[d, "{(\alpha, f)}"'] \arrow[r, phantom, "\mapsto"] & F.A \arrow[d, "G.f \circ \alpha_A"] \\
(G, B) & G.B
\end{tikzcd}$$

Note that we could have chosen $\alpha_B \circ F.f$ instead of $G.f \circ \alpha_A$, but they are equal because α is a natural transformation.

We will sometimes *section* @ for a particular A in $\mathbb{C}$ to get a unary functor $(@A) : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}$ that "evaluates at A":

$$\begin{tikzcd}
(@A) : \mathbb{D}^{\mathbb{C}} \arrow[r] & \mathbb{C} \\
F \arrow[d, "\alpha"'] \arrow[r, phantom, "\mapsto"] & F.A \arrow[d, "\alpha_A"] \\
G & G.A
\end{tikzcd}$$

We can think of a functor $F : \mathbb{C} \to \mathbb{D}$ as defining a family of $\mathbb{D}$-objects indexed by the objects of $\mathbb{C}$; then applying $(@A)$ to F selects the A-th member of that family:

$$(@A).F = F.A$$

Similarly, applying $(@A)$ to a natural transformation $\alpha : F \mathrel{\dot{\to}} G$ selects the A-th instance of α:

$$(@A).\alpha = \alpha_A : F.A \to G.A$$

3.1.2. Functor lifting

We now describe how to *lift* functors to operate between functor categories. Given a functor $F : \mathbb{C} \to \mathbb{D}$, we can extend it to a functor between functor categories:

$$\dot{F} : \mathbb{C}^{\mathbb{B}} \longrightarrow \mathbb{D}^{\mathbb{B}}$$

$$\begin{tikzcd} G \arrow[d, "\alpha"'] & FG \arrow[d, "F\alpha"] \\ H \arrow[r, phantom, "\mapsto" pos=0.5, shift left=4ex] & FH \end{tikzcd}$$

The functor $\dot{F}$ simply postcomposes F with its argument. An alternative notation for the lifting of F might be as a sectioned composition operator $(F\circ) : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$, where the composition symbol here denotes composition of functors. We prefer $\dot{F}$ however, because it leads to more readable expressions, particularly when dealing with lifted bifunctors.

If objects and arrows of $\mathbb{C}^{\mathbb{B}}$ are viewed as families of $\mathbb{C}$-objects and $\mathbb{C}$-arrows indexed by the objects of $\mathbb{B}$, then the behaviour of $\dot{F}$ when applied to one of these families is to apply F to every member of the family, thus turning it into a family of $\mathbb{D}$-objects or $\mathbb{D}$-arrows. We can see this by *selecting* a single member of the resulting family using the "evaluate at A" functor for some A in $\mathbb{B}$:

$$\begin{array}{ll} & (@A).(\dot{F}.G) \\ = & \quad \{ \text{definition of } \dot{F} \} \\ & (@A).(FG) \\ = & \quad \{ \text{definition of } (@A) \} \\ & FG.A \\ = & \quad \{ \text{functor composition} \} \\ & F.(G.A) \end{array}$$

$=\quad$ { definition of $(@A)$ }

$F.((@A).G)$

where G is any object in $\mathbb{C}^{\mathbb{B}}$. This shows that the A-th member of $\dot{F}.G$ is F applied to the A-th member of G. We can prove the same for arrows of $\mathbb{C}^{\mathbb{B}}$, and this gives us the following lemma:

Lemma 3.1: For any functor $F : \mathbb{C} \to \mathbb{D}$ and object A in $\mathbb{B}$, the following diagram commutes:

$$\begin{tikzcd}
\mathbb{C}^{\mathbb{B}} \arrow[r, "\dot{F}"] \arrow[d, "(@A)"'] & \mathbb{D}^{\mathbb{B}} \arrow[d, "(@A)"] \\
\mathbb{C} \arrow[r, "F"'] & \mathbb{D}
\end{tikzcd}$$

that is, $F(@A) = (@A)\dot{F}$.

When lifting *bifunctors* or *n-ary functors* we must take care. Let $\otimes : \mathbb{C} \times \mathbb{D} \to \mathbb{E}$ be a bifunctor. If we follow the above lifting process directly then we end up with a functor

$$\dot{\otimes} : (\mathbb{C} \times \mathbb{D})^{\mathbb{B}} \to \mathbb{E}^{\mathbb{B}}$$

which, although a lifted functor, is no longer a bifunctor. Fortunately we have the following isomorphism:

Lemma 3.2: For any categories $\mathbb{C}$ and $\mathbb{D}$ and *small* category $\mathbb{B}$, there is an isomorphism

$$\mathbb{C}^{\mathbb{B}} \times \mathbb{D}^{\mathbb{B}} \cong (\mathbb{C} \times \mathbb{D})^{\mathbb{B}}$$

Proof: Immediate from the fact that **Cat** has products.

Then we can construct our lifted bifunctor by precomposing the isomorphism:

$$\dot{\otimes} : \mathbb{C}^{\mathbb{B}} \times \mathbb{D}^{\mathbb{B}} \xrightarrow{\cong} (\mathbb{C} \times \mathbb{D})^{\mathbb{B}} \longrightarrow \mathbb{E}^{\mathbb{B}}$$

It is straightforward to check that for any $F : \mathbb{B} \to \mathbb{C}$ and $G : \mathbb{B} \to \mathbb{D}$, and any object A in $\mathbb{B}$ that

$$(F \mathbin{\dot{\otimes}} G).A = (F.A) \otimes (G.A)$$

and this agrees with the description of functor lifting we gave in section 2.1. The action of $\dot{\otimes}$ on natural transformations $\alpha : F \mathrel{\underset{\bullet}{\longrightarrow}} G$ and $\beta : H \mathrel{\underset{\bullet}{\longrightarrow}} K$ is to produce a new natural transformation $\alpha \mathbin{\dot{\otimes}} \beta : F \mathbin{\dot{\otimes}} H \mathrel{\underset{\bullet}{\longrightarrow}} G \mathbin{\dot{\otimes}} K$, with components

$$(\alpha \mathbin{\dot{\otimes}} \beta)_A = \alpha_A \otimes \beta_A$$

for each object A in $\mathbb{B}$.

3.1.3. Lifting other operations

Functors can be lifted pointwise to operate between functor categories, and functors sometimes have related operations that can also be lifted. Take, for example, a category $\mathbb{C}$ that has binary sums

$$+ : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$$

The sum functor can be lifted for any small category $\mathbb{B}$ to give a functor

$$\dot{+} : \mathbb{C}^{\mathbb{B}} \times \mathbb{C}^{\mathbb{B}} \to \mathbb{C}^{\mathbb{B}}$$

This lifted functor $\dot{+}$ gives categorical sums in the functor category $\mathbb{C}^{\mathbb{B}}$, and the associated *join* operation, $\dot{\triangledown}$, is computed pointwise from the join operation for $+$. That is, for any functors $F, G, H : \mathbb{B} \to \mathbb{C}$ and natural transformations $\alpha : F \mathrel{\underset{\bullet}{\longrightarrow}} H$ and $\beta : G \mathrel{\underset{\bullet}{\longrightarrow}} H$, we can construct a natural transformation

$$\alpha \mathbin{\dot{\triangledown}} \beta : F \mathbin{\dot{+}} G \mathrel{\underset{\bullet}{\longrightarrow}} H$$

defined by

$$(\alpha \mathbin{\dot{\triangledown}} \beta)_A \overset{def}{=} \alpha_A \triangledown \beta_A$$

for any A in $\mathbb{B}$. The same can be done for products and the associated *split* operation for products.

3.2. Initial algebras in functor categories

In order to talk about algebras in functor categories we need to use functors between functor categories — that is, functors that operate on functors and natural transformations. Bird and Meertens [BirdM98] call these *higher-order functors* or *hofunctors*. We have seen examples of such functors already — the *lifted* functors we defined in section 3.1.2. In this section we show how the *polymorphic* constructors of a parameterized data-type form initial algebras for higher-order functors.

Consider the data-type of parameterized lists — this has polymorphic constructors with components for each type A in our base category $\mathbb{C}$:

$$\mathsf{NilL}_A : \mathbb{1} \to \mathit{List}.A$$

$$\mathsf{ConsL}_A : A \times \mathit{List}.A \to \mathit{List}.A$$

where *List* is the type functor that arises from the standard list base bifunctor. Another way of writing this is that the constructors are natural transformations:

$$\mathsf{NilL} : \underline{\mathbb{1}} \mathrel{\underset{\bullet}{\longrightarrow}} \mathit{List}$$

$$\mathsf{ConsL} : \mathit{Id}_{\mathbb{C}} \mathbin{\dot{\times}} \mathit{List} \mathrel{\underset{\bullet}{\longrightarrow}} \mathit{List}$$

And it is possible to join these pointwise into a single arrow:

$$\begin{tikzcd}
\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} List \arrow[d, "\mathsf{NilL} \mathbin{\dot{\triangledown}} \mathsf{ConsL}"] \\
List
\end{tikzcd}$$

We want this to be an algebra for some higher-order functor F. The type functor $List : \mathbb{C} \to \mathbb{C}$ will be the carrier of this algebra, so we want F to map $List$ to $\underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} List$, which is also an endofunctor on $\mathbb{C}$. The obvious definition of F is

$$\begin{tikzcd}
F : \mathbb{C}^{\mathbb{C}} \arrow[r] & \mathbb{C}^{\mathbb{C}} \\
H \arrow[d, "\alpha"'] & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} H \arrow[d, "Id \dot{+} Id \dot{\times} \alpha"] \\
K \arrow[r, phantom, "\mapsto", near start, shift left=4ex] & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} K
\end{tikzcd}$$

F is a higher-order functor that maps endofunctors on $\mathbb{C}$ (objects of $\mathbb{C}^{\mathbb{C}}$) to other endofunctors on $\mathbb{C}$. Now that we have F, it is easy to see that the polymorphic constructors form an F-algebra:

$$\mathsf{NilL} \mathbin{\dot{\triangledown}} \mathsf{ConsL} : F.List \to List$$

The question is whether this algebra is *initial* or not. Fortunately we can state a general result that relates "higher-order" initial algebras to any parameterized data-types that arise as initial algebras using the bifunctor construction described in section 2.3:

Theorem 3.1:

Let $\mathbb{C}$ be a category and $\ddagger : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ a bifunctor. If, for every object A in $\mathbb{C}$, the sectioned functor $(A\ddagger) : \mathbb{C} \to \mathbb{C}$ has an initial algebra,

$$in^{A\ddagger} : A \ddagger \mu(A\ddagger) \to \mu(A\ddagger)$$

Then these initial algebras form the components of an initial algebra of the higher-order functor $(Id_{\mathbb{C}}\dot{\ddagger}) : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}^{\mathbb{C}}$ with the type functor $\textcircled{\ddagger}$ as the carrier:

$$in^{Id_{\mathbb{C}}\dot{\ddagger}} : Id_{\mathbb{C}} \dot{\ddagger} \textcircled{\ddagger} \mathrel{\dot{\to}} \textcircled{\ddagger}$$

$$(in^{Id_{\mathbb{C}}\ddagger})_A \overset{def}{=} in^{A\ddagger}$$

Proof: In appendix A.1.

In particular, this includes all *regular* parameterized types. Therefore, as well as being initial algebras of sectioned bifunctors, regular types are also initial algebras of higher-order functors.

How does working with higher-order functors help us? We have seen that non-uniform data-types cannot be expressed as initial algebras of sectioned bifunctors because the action of sectioning corresponds to fixing the parameter of the data-type. However, linear non-uniform data-types *can* be expressed as initial algebras of higher-order functors, and the “higher-order” catamorphisms that arise from these initial algebras are folds on the non-uniform data-types. We demonstrate this for nests, and then show how to derive folds for other linear non-uniform data-types.

3.2.1. Polymorphic folds for nests

Let us consider the *Nest* data-type. The polymorphic constructors would have types

$$\underline{\mathbb{1}} \underset{\cdot}{\longrightarrow} Nest \qquad\qquad Id_{\mathbb{C}} \dot{\times} (Nest)P \underset{\cdot}{\longrightarrow} Nest$$

If we follow the same reasoning as we did for lists then we arrive at the higher-order base functor

$$\begin{tikzcd}[row sep=small]
N : \mathbb{C}^{\mathbb{C}} \arrow[rr] & & \mathbb{C}^{\mathbb{C}} \\
H \arrow[dd, "\alpha"'] & & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} HP \arrow[dd, "Id \dot{+} Id \dot{\times} \alpha P"] \\
 & \mapsto & \\
K & & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} KP
\end{tikzcd}$$

Let us assume for now that N has an initial algebra $(Nest, in^N)$ — we will look at the problem of proving the existence of initial algebras later in this chapter. We can decompose in^N into two polymorphic constructors, say NilN and ConsN:

$$in^N = \mathsf{NilN} \mathbin{\dot{\triangledown}} \mathsf{ConsN} : N.Nest \mathrel{\dot{\to}} Nest$$

From "higher-order" initial algebras we get "higher-order" catamorphisms. These follow exactly the same construction as normal catamorphisms but they work at the level of polymorphic functions rather than monomorphic ones. Explicitly, given an N-algebra $\alpha : N.R \mathrel{\dot{\to}} R$ for some endofunctor R, then we have a catamorphism $\llparenthesis \alpha \rrparenthesis : Nest \mathrel{\dot{\to}} R$ that uniquely satisfies

$$\begin{tikzcd}
\underline{\mathbb{1}} \mathbin{\dot{+}} Id_{\mathbb{C}} \mathbin{\dot{\times}} (Nest)P \arrow[r, "Id \mathbin{\dot{+}} Id \mathbin{\dot{\times}} \llparenthesis \alpha \rrparenthesis P"] \arrow[d, "\mathsf{NilN} \mathbin{\dot{\triangledown}} \mathsf{ConsN}"'] & \underline{\mathbb{1}} \mathbin{\dot{+}} Id_{\mathbb{C}} \mathbin{\dot{\times}} RP \arrow[d, "\alpha"] \\
Nest \arrow[r, "\llparenthesis \alpha \rrparenthesis"'] & R
\end{tikzcd}$$

If we decompose α into the pointwise join of two natural transformations

$$\lambda : \underline{\mathbb{1}} \mathrel{\dot{\to}} R \qquad\qquad \rho : Id_{\mathbb{C}} \mathbin{\dot{\times}} RP \mathrel{\dot{\to}} R$$

then we can write the above diagram as the defining equations:

$$\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis \circ \mathsf{NilN} = \lambda \circ Id = \lambda \tag{3.2.1}$$

$$\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis \circ \mathsf{ConsN} = \rho \circ (Id \mathbin{\dot{\times}} \llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis P) \tag{3.2.2}$$

The meaning of $\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis P$ may require some explanation. In this higher-order setting, catamorphisms are natural transformations, and for each A in $\mathbb{C}$,

$$\begin{aligned}
& (\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis P)_A \\
= \quad & \{\text{ composition of functors and natural transformations }\} \\
& \llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis_{P.A}
\end{aligned}$$

So the precomposition of the P functor to the catamorphism has the effect of applying the P functor to the type at which the polymorphic function is to be instantiated, before it is instantiated. So, for each A in $\mathbb{C}$, $\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis_A$ has type

$$\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis_A : Nest.A \to R.A$$

and $(\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis P)_A$ has type

$$(\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis P)_A : Nest.P.A \to R.P.A$$

It is in this way that the recursive call to $\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis$ in the above definition is coerced to be of the correct type to match the changing structure of the nest data-type.

We can easily turn the above recursion pattern into a Haskell program — following the point-free defining equations 3.2.1 and 3.2.2 we get the pointwise definition of the function:

```
foldNest lam rho NilN = lam
foldNest lam rho (ConsN x xs) = rho x (foldNest lam rho xs)
```

The Haskell type system can work out that the above coercion using P is necessary, and there is no need for us to indicate it in our program. However, the coercion in the recursive call means that we have made use of polymorphic recursion and must therefore supply a type signature. An important difference between this and the previous folds is that the arguments lam and rho must be *polymorphic* functions. This requires the use of rank-2 type signatures, and the correct type signature is:

```
foldNest :: (∀ a . r a) →
                 (∀ a . a → r (P a) → r a) →
                     Nest b → r b
```

where r is the type constructor that is the carrier of the algebra given by the arguments.

Let us see how this fold operator can be used in action. One of the simplest applications of catamorphisms on nests is the catamorphism that collapses the pairing inside the nests and thus “flattens” a nest into a list. This example is essentially the same as Bird and Meerten’s “listify” example [BirdM98, section 5]. To construct the appropriate catamorphism we need an auxiliary function that turns a list of pairs of A’s into a list of A’s by simply “unpacking” the pairs:

$$\gamma : (\mathit{List})P \xrightarrow[\bullet]{} \mathit{List}$$

In other words, γ would map the list $[(1,2),(3,4),(5,6)]$ to the list $[1,2,3,4,5,6]$. This γ performs the collapsing of the pairing inside the nest. The definition of γ need not concern us here.

Once we have γ, we can combine it with the list constructors to form an N-algebra:

$$\begin{tikzcd}
\underline{\mathbb{1}} \dot{+} \mathit{Id}_{\mathbb{C}} \dot{\times} (\mathit{List})P \arrow[d, "\mathit{Id} \dot{+} \mathit{Id} \dot{\times} \gamma"] \\
\underline{\mathbb{1}} \dot{+} \mathit{Id}_{\mathbb{C}} \dot{\times} \mathit{List} \arrow[d, "\mathsf{NilL} \dot{\triangledown} \mathsf{ConsL}"] \\
\mathit{List}
\end{tikzcd}$$

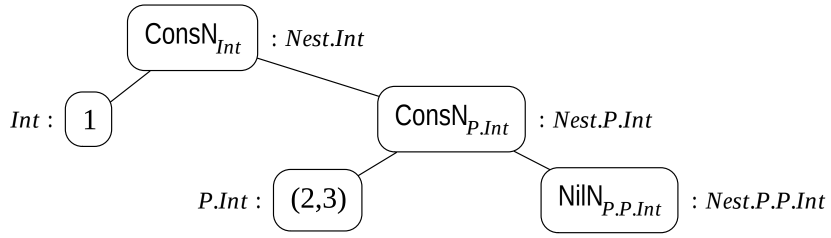
which we can rewrite as

$$\begin{array}{ll}
 & (\mathsf{NilL} \dot{\triangledown} \mathsf{ConsL}) \circ (\mathit{Id} \dot{+} \mathit{Id} \dot{\times} \gamma) \\
= & \qquad \{\text{ join fusion }\} \\
 & (\mathsf{NilL} \circ \mathit{Id}) \dot{\triangledown} (\mathsf{ConsL} \circ (\mathit{Id} \dot{\times} \gamma)) \\
= & \qquad \{\text{ identity }\} \\
 & \mathsf{NilL} \dot{\triangledown} (\mathsf{ConsL} \circ (\mathit{Id} \dot{\times} \gamma))
\end{array}$$

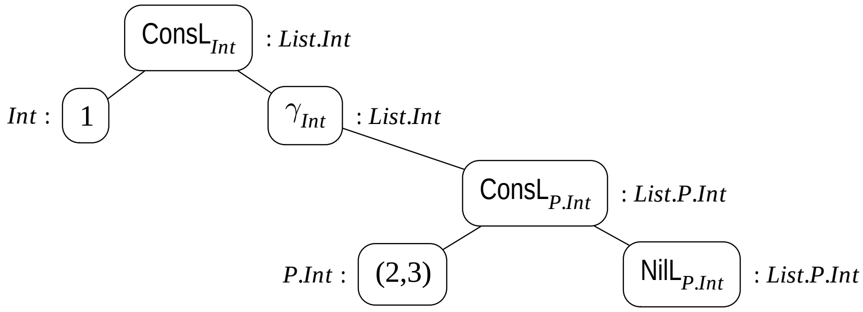
This algebra can be given as the argument to our higher-order catamorphism operator to produce the catamorphism:

$$(\!|\, \mathsf{NilL} \dot{\triangledown} (\mathsf{ConsL} \circ (\mathit{Id} \dot{\times} \gamma)) \,|\!) : \mathit{Nest} \xrightarrow[\bullet]{} \mathit{List}$$

We can see the effect on a simple nest of integers:



after applying the catamorphism and simplifying, this becomes



which gives the list [1, 2, 3]

3.2.2. Polymorphic folds for other non-uniform Haskell data-types

We now show how to construct catamorphisms for *linear* non-uniform data-types given by Haskell type definitions of the form

$$\begin{array}{l} \underline{\mathsf{data}}\ \mathsf{T}\ \mathsf{a} = \mathsf{C}^1\ \mathsf{E}_1 \\ \qquad\qquad \ldots \\ \qquad\quad \mid \mathsf{C}^n\ \mathsf{E}_n \end{array}$$

where n is the number of constructors, each C^i is the name of the i-th constructor, and each E_i describes the source type of the i-th constructor. We restrict our attention to simple sum-of-product data-types, so we do not allow function spaces to appear in the type definition.

In a *uniform* data-type, the E_i's can depend on the parameter a, and they can also refer recursively to (T a) — the type being defined. Consider the uniform type of parameterized cons-lists

```
data List a = NilL
            | ConsL a (List a)
```

The first constructor, C^1, is NilL, and its corresponding E_1 is empty. The second constructor, C^2, is ConsL, and E_2 is the product of the parameter a and a recursive call to (List a).

For *non-uniform* types, we must also admit the possibility that the E_i's can refer non-uniformly to the type being defined, that is, to (T $(\mathsf{M}_1$ a)),...,(T $(\mathsf{M}_m$ a)) for some modifying functors $\mathsf{M}_1,\ldots,\mathsf{M}_m$. We can show the dependencies as

$$\begin{aligned}\underline{\mathsf{data}}\ \mathsf{T\ a} = {} & \mathsf{C}^1\ \mathsf{E}_1(\mathsf{a,\ T\ a,\ T\ (M_1\ a),\ldots,\ T\ (M_m\ a)})\\ & \ldots\\ \mid\ & \mathsf{C}^n\ \mathsf{E}_n(\mathsf{a,\ T\ a,\ T\ (M_1\ a),\ldots,\ T\ (M_m\ a)})\end{aligned}$$

Consider nests:

```
data Nest a = NilN
            | ConsN a (Nest (P a))
```

The first constructor, C^1, is NilN, and E_1 is empty. The second constructor, C^2, is ConsN, and E_2 is the product of the parameter a and the recursive call (Nest (P a)). The type uses only a single modifying functor, so M_1 is P.

Categorically, for a type definition of the above form, we have constructor instances:

$$\mathsf{C}^i_A : E_i(A, T.A, TM_1.A, \ldots, TM_m.A) \to T.A$$

for each object A in $\mathbb{C}$ and $1 \leq i \leq n$. If we *lift* the E_i's to operate on functors so that

$$E_i(A, T.A, TM_1.A, \ldots, TM_m.A) = \dot{E}_i(Id, T, TM_1, \ldots, TM_m).A$$

then the polymorphic constructors can be viewed as natural transformations

$$\mathsf{C}^i : \dot{E}_i(Id, T, TM_1, \ldots, TM_m) \xrightarrow[\bullet]{} T$$

Continuing with the nest example, the constructor NilN is a constant, so $\dot{E}_1$ ignores all its arguments:

$$\dot{E}_1(Id, T, TP).A = \mathbb{1}$$

for any object A in $\mathbb{C}$. Or equivalently, at the functor level,

$$\dot{E}_1(Id, T, TP) = \underline{\mathbb{1}}$$

For ConsN we have

$$\dot{E}_2(Id, T, TP).A = A \times TP.A$$

or at the functor level,

$$\dot{E}_2(Id, T, TP) = Id \mathbin{\dot{\times}} TP$$

and so the constructors are natural transformations

$$\mathsf{NilN} : \dot{E}_1(Id, T, TP) \xrightarrow[\bullet]{} T$$

$$\mathsf{ConsN} : \dot{E}_2(Id, T, TP) \xrightarrow[\bullet]{} T$$

Once we have the source functors for our polymorphic constructors, we can collect them into an algebra:

$$\begin{tikzcd} \dot{E}_1(Id, T, TM_1, \ldots, TM_m) \mathbin{\dot{+}} \cdots \mathbin{\dot{+}} \dot{E}_n(Id, T, TM_1, \ldots, TM_m) \arrow[d, "\mathsf{C}^1 \mathbin{\dot{\triangledown}} \cdots \mathbin{\dot{\triangledown}} \mathsf{C}^n"] \\ T \end{tikzcd}$$

The base functor for the algebra is a higher-order functor

$$\begin{tikzcd}
B : \mathbb{C}^{\mathbb{C}} \arrow[r] & \mathbb{C}^{\mathbb{C}} \\
X \arrow[d, "\alpha"'] \arrow[r, phantom, "\mapsto"] & \dot{E}_1(Id, X, XM_1, \ldots, XM_m) \dot{+} \cdots \dot{+} \dot{E}_n(Id, X, XM_1, \ldots, XM_m) \arrow[d, "\dot{E}_1(Id, \alpha, \alpha M_1, \ldots, \alpha M_m) \dot{+} \cdots \dot{+} \dot{E}_n(Id, \alpha, \alpha M_1, \ldots, \alpha M_m)"] \\
Y & \dot{E}_1(Id, Y, YM_1, \ldots, YM_m) \dot{+} \cdots \dot{+} \dot{E}_n(Id, Y, YM_1, \ldots, YM_m)
\end{tikzcd}$$

and $\mathsf{C}^1 \dot{\triangledown} \cdots \dot{\triangledown} \mathsf{C}^n$ is a B-algebra with carrier T. In fact, $\mathsf{C}^1 \dot{\triangledown} \cdots \dot{\triangledown} \mathsf{C}^n$ is an *initial* (strict) B-algebra. We have seen how the fold combinator for nests is derived from the catamorphism diagram for the initial algebra $\mathsf{NilN} \dot{\triangledown} \mathsf{ConsN}$. Fold combinators for other types can be derived in the same way — for an initial algebra $\mathsf{C}^1 \dot{\triangledown} \cdots \dot{\triangledown} \mathsf{C}^n$ formed out of the constructors C^1 to C^n, the catamorphism diagram:

$$\begin{tikzcd}
B.T \arrow[r, "B.(\!|\gamma|\!)"] \arrow[d, "\mathsf{C}^1 \dot{\triangledown} \cdots \dot{\triangledown} \mathsf{C}^n"'] & B.R \arrow[d, "\gamma^1 \dot{\triangledown} \cdots \dot{\triangledown} \gamma^n"] \\
T \arrow[r, dotted, "(\!|\gamma|\!)"'] & R
\end{tikzcd}$$

for some $\gamma^1, \ldots, \gamma^n$ captures the n defining equations:

$$(\!|\gamma|\!) \circ \mathsf{C}^i = \gamma^i \circ \dot{E}_i(Id, (\!|\gamma|\!), (\!|\gamma|\!)M_1, \ldots, (\!|\gamma|\!)M_m)$$

for $1 \leq i \leq n$. Translating these defining equations into pointwise Haskell definitions gives the desired fold combinator.

3.3. Examples and non-examples

We now give some examples of the sort of computations that can be performed using higher-order catamorphisms, and also some that can't. Remember that the catamorphisms we are defining are actually natural transformations, and so only operations that are parametrically polymorphic stand a chance of being able to be expressed as a higher-order catamorphism. This excludes many useful non-polymorphic functions, such as summing a nest of integers, which we might reasonably expect to be expressible as a fold. Furthermore, the same reasoning shows us that we cannot express maps over non-uniform data-types as higher-order catamorphisms. This is a major blow because the properties of map are traditionally proved in Squiggol by appealing to the induction principle they receive by being expressed as catamorphisms.

We have already seen one example of a higher-order catamorphism used to flatten nests into lists. Now we give a slightly unusual example for monads in which the actual non-uniform data-type is inspired by the signature of the polymorphic monad operations.

3.3.1. Monad normalizing

A categorical structure that has proved to have many applications in computer science is that of *monad* [MacLane71]:

> **Definition 3.3 (monad):** A *monad* $\left\langle M, \eta, \mu \right\rangle$ consists of an endofunctor $M : \mathbb{C} \to \mathbb{C}$ and two natural transformations,
>
> $$\eta : Id_{\mathbb{C}} \mathrel{\dot{\longrightarrow}} M \qquad\qquad \mu : MM \mathrel{\dot{\longrightarrow}} M$$
>
> called respectively the *unit* and the *join* of the monad, that satisfy the following laws:
>
> $$\begin{tikzcd} & MMM \arrow[ld, "M\mu"'] \arrow[rd, "\mu M"] & \\ MM \arrow[rd, "\mu"'] & & MM \arrow[ld, "\mu"] \\ & M & \end{tikzcd} \qquad \begin{tikzcd} M \arrow[r, "M\eta"] \arrow[rd, "Id"'] & MM \arrow[d, "\mu"] & M \arrow[l, "\eta M"'] \arrow[ld, "Id"] \\ & M & \end{tikzcd}$$

Wadler [Wadler92] gives examples of how monads can be used in functional programming.

The monad operations *unit* and *join* naturally form a higher-order algebra:

$$\eta \mathbin{\dot{\triangledown}} \mu : Id_{\mathbb{C}} \mathbin{\dot{+}} MM \underset{\bullet}{\longrightarrow} M$$

and a suitable base functor for this algebra can easily be found:

$$\begin{tikzcd}
V : \mathbb{C}^{\mathbb{C}} \arrow[r] & \mathbb{C}^{\mathbb{C}} \\
F \arrow[d, "\alpha"'] \arrow[r, phantom, "\mapsto"] & Id_{\mathbb{C}} \dot{+} FM \arrow[d, "Id_{\mathbb{C}} \dot{+} \alpha M"] \\
G & Id_{\mathbb{C}} \dot{+} GM
\end{tikzcd}$$

Suppose this functor has an initial algebra

$$in^{V} = \mathsf{ZeroV} \mathbin{\dot{\triangledown}} \mathsf{SuccV} : Id_{\mathbb{C}} \mathbin{\dot{+}} (\mu V)M \to \mu V$$

which would correspond to the Haskell data-type definition

```
data MuV a = ZeroV a
           | SuccV (MuV (M a))
```

This data-type can store values of type a, (M a), (M (M a)), and so on. The number of applications of the M functor can be found by counting the SuccV constructors. If we take M to be the list type constructor then some example values might be

```
(ZeroV 6) :: MuV Int

(SuccV (ZeroV [1,2,3])) :: MuV Int

(SuccV (SuccV (ZeroV [[1,2], [], [36,12]]))) :: MuV Int
```

where, for convenience, we have used Haskell's built-in square-bracket notation for lists. We can see that for each additional SuccV constructor, the depth of the nesting of the lists increases by one.

For any V-algebra, say,

$$\lambda \mathbin{\dot{\triangledown}} \rho : Id_{\mathbb{C}} \dotplus RM \underset{\bullet}{\longrightarrow} R$$

we can construct the higher-order catamorphism:

$$\begin{tikzcd}
Id_{\mathbb{C}} \dotplus (\mu V)M \arrow[r, "Id \dotplus \llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis M"] \arrow[d, "\mathsf{ZeroV} \mathbin{\dot{\triangledown}} \mathsf{SuccV}"'] & Id_{\mathbb{C}} \dotplus RM \arrow[d, "\lambda \mathbin{\dot{\triangledown}} \rho"] \\
\mu V \arrow[r, dotted, "\llparenthesis \lambda \mathbin{\dot{\triangledown}} \rho \rrparenthesis"'] & R
\end{tikzcd}$$

from which we can define the combinator

```
foldMuV :: (∀ a . a → r a) →
               (∀ a . r (M a) → r a) →
                  MuV b → r b
foldMuV lam rho (ZeroV x) = lam x
foldMuV lam rho (SuccV x) = rho (foldMuV lam rho) x
```

Now we can consider the effects of using the monad operations $\eta \mathbin{\dot{\triangledown}} \mu$ as the argument algebra. We get a catamorphism:

$$\llparenthesis \eta \mathbin{\dot{\triangledown}} \mu \rrparenthesis : \mu V \underset{\bullet}{\longrightarrow} M$$

The action of $\llparenthesis \eta \mathbin{\dot{\triangledown}} \mu \rrparenthesis$ is to "normalize" values of type μV so that they are of a single application of the M functor.

For example, let M be the list monad with the singleton list constructor

$$\mathsf{singleton} : Id_{\mathbb{C}} \underset{\bullet}{\longrightarrow} List$$

as the unit operation and the function that concatenates a list of lists into a single list,

$$\mathsf{concat} : (List)(List) \underset{\bullet}{\longrightarrow} List$$

as the join operation. Then the catamorphism

$$(\!|\,\mathsf{singleton} \mathbin{\dot{\triangledown}} \mathsf{concat}\,|\!) : \mu V \underset{\bullet}{\longrightarrow} List$$

has the effect of collapsing nested lists of lists into a single list, and it wraps non-list values into a singleton list using the singleton constructor.

3.3.2. Summing a nest and mapping non-examples

A typical operation we might like to define on nests of numbers is a function to compute the sum of all the numbers in a nest. Suppose the numbers are integers, then we desire a function

$$\mathsf{sumNest} : Nest.Int \to Int$$

Let $\underline{Int}$ be the constant functor that always returns integers, then we could try to find a catamorphism

$$\begin{tikzcd}
\underline{\mathbb{1}} \mathbin{\dot{+}} Id_{\mathbb{C}} \mathbin{\dot{\times}} (Nest)P \arrow[r, "Id \dot{+} Id \dot{\times} (\!|\lambda \dot{\triangledown} \rho|\!)P"] \arrow[d, "\mathsf{NilN} \dot{\triangledown} \mathsf{ConsN}"'] & \underline{\mathbb{1}} \mathbin{\dot{+}} Id_{\mathbb{C}} \mathbin{\dot{\times}} (\underline{Int})P \arrow[d, "\lambda \dot{\triangledown} \rho"] \\
Nest \arrow[r, dotted, "(\!|\lambda \dot{\triangledown} \rho|\!)"'] & \underline{Int}
\end{tikzcd}$$

for some λ and ρ. The λ case is easy — both source and target functors are constant, so each component has the same type:

$$\lambda_A : \mathbb{1} \to Int$$

and we can set this to be the integer zero, as it will be the sum of empty nests. However, when we try to find the appropriate ρ, the source functor is not constant, and we are forced to provide, for each A in $\mathbb{C}$, an arrow:

$$\rho_A : A \times \mathit{Int} \to \mathit{Int}$$

In the case when A is Int, we can set ρ_{Int} to be the integer addition function, but the problem lies in what to do when A is not Int — there is no sensible way to define the rest of the ρ_A's so that ρ is a natural transformation. This is because summing integers is not a parametrically polymorphic operation — it relies on the parameter being Int. Therefore we cannot construct a parametric polymorphic algebra $\lambda \mathbin{\dot{\triangledown}} \rho$ for which the catamorphism $\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis$ will sum nests of integers.

The same problem manifests itself when we try to express maps as higher-order catamorphisms. Given a function $f : A \to B$, we would like to construct the corresponding map

$$\mathsf{mapNestf} : \mathit{Nest}.A \to \mathit{Nest}.B$$

using a higher-order catamorphism, in particular, as the A-th instance of a catamorphism:

$$\llparenthesis\, \gamma \,\rrparenthesis_A : \mathit{Nest}.A \to R.A$$

for some $\gamma : N.R \to R$. Again the problem is parametric polymorphism — using the function f we can construct a single instance of the desired algebra γ:

$$\begin{tikzcd}
\mathbb{1} + A \times \mathit{Nest}.P.B \arrow[d, "\mathit{Id} + f \times \mathit{Id}"] \\
\mathbb{1} + B \times \mathit{Nest}.P.B \arrow[d, "\mathsf{NilN}_B \triangledown \mathsf{ConsN}_B"] \\
\mathit{Nest}.B
\end{tikzcd}$$

but we cannot, in general, extend this single instance to a parametric polymorphic algebra.

3.4. Existence of initial algebras in functor categories

We have seen that regular parameterized types give rise to initial algebras in functor categories, but this does not guarantee the existence of initial algebras for non-uniform types. Are there general conditions for the existence of initial algebras in a functor category? In this section we will look at transferring the results from section 2.5 about the existence of initial algebras to operate in the setting of functor categories.

One of the nice properties of functor categories is that they inherit much of the structure of their target category. Importantly, they inherit all the completeness properties of their target category, and they do so in a pointwise manner:

Theorem 3.2 (colimits in functor categories are computed pointwise): Let $\mathbb{D}$ be a category and $\mathbb{C}$ and $\mathbb{J}$ be small categories. Let $D : \mathbb{J} \to \mathbb{D}^{\mathbb{C}}$ be a $\mathbb{J}$-diagram in $\mathbb{D}^{\mathbb{C}}$. If, for each A in $\mathbb{C}$, the diagram

$$\mathbb{J} \xrightarrow{D} \mathbb{D}^{\mathbb{C}} \xrightarrow{(@A)} \mathbb{D}$$

has a colimiting cocone in $\mathbb{D}$, then D has a colimiting cocone in $\mathbb{D}^{\mathbb{C}}$. In particular, the colimiting object of D, $Col(D) : \mathbb{C} \to \mathbb{D}$, is defined pointwise from the colimiting objects of the $(@A)D$ diagrams, so

$$(@A).Col(D) = Col((@A)D)$$

Proof: See Mac Lane's section on *limits with parameters* [MacLane71, section V.3] or Schubert's section on *limits in functor categories* [Schubert72, sections 7.5 and 8.5].

The same can be shown for limits. This means that if all the pointwise limits or colimits exist then "evaluate at A" functors preserve limits or colimits:

Corollary 3.1: If $\mathbb{D}$ is $\mathbb{J}$-(co)complete for some small $\mathbb{J}$ and $\mathbb{C}$ is some small non-empty category then for each A in $\mathbb{C}$, $(@A) : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}$ is $\mathbb{J}$-(co)continuous.

Proof: See [Schubert72, section 7.5.4] or [MacLane71, section V.3].

It also means that if the target category has all $\mathbb{J}$-limits or $\mathbb{J}$-colimits then so does the functor category:

Corollary 3.2 (functor categories inherit completeness from their target category): Let $\mathbb{C}$ be a small category and $\mathbb{D}$ be an arbitrary category. If $\mathbb{D}$ is $\mathbb{J}$-complete or $\mathbb{J}$-cocomplete for some small category $\mathbb{J}$, then so is $\mathbb{D}^{\mathbb{C}}$.

Proof: Let $D : \mathbb{J} \to \mathbb{D}^{\mathbb{C}}$ be some $\mathbb{J}$-diagram. Because $\mathbb{D}$ is $\mathbb{J}$-complete or $\mathbb{J}$-cocomplete, we know that all the limits or colimits exist for the diagrams $(@A)D : \mathbb{J} \to \mathbb{D}$ for each A in $\mathbb{C}$. Then, by theorem 3.2 or its dual, the limit or colimit of D must exist.

The particular instances that we will be interested in are summarized as follows:

Corollary 3.3: For any category $\mathbb{D}$ and small category C,

(i) if $\mathbb{D}$ has an initial or terminal object then so does $\mathbb{D}^{\mathbb{C}}$;

(ii) if $\mathbb{D}$ has products or sums then so does $\mathbb{D}^{\mathbb{C}}$;

(iii) if $\mathbb{D}$ is ω-cocomplete then so is $\mathbb{D}^{\mathbb{C}}$;

(iv) if $\mathbb{D}$ is ω^{op}-complete then so is $\mathbb{D}^{\mathbb{C}}$;

and these immediately give us a version of the generalized Kleene fixed-point theorem (theorem 2.2) for functor categories:

Corollary 3.4: If $\mathbb{D}$ is an ω-cocomplete category with an initial object, $\mathbb{C}$ is a small category and $F : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}^{\mathbb{C}}$ is an ω-cocontinuous higher-order endofunctor then F has an initial algebra.

Proof: By corollary 3.3 we know that $\mathbb{D}^{\mathbb{C}}$ is ω-cocomplete and has an initial object, then simply apply the generalized Kleene fixed-point theorem.

All we have done is to reduce the completeness conditions on $\mathbb{D}^{\mathbb{C}}$ to completeness conditions on the base category $\mathbb{D}$.

3.4.1. Proving continuity properties of higher-order functors

In order to use corollary 3.4 to prove the existence of initial algebras for a non-uniform type, we must show that the base functor for the non-uniform type is ω-cocontinuous. Let us look at the base functor for nests:

$$\begin{tikzcd}
N : \mathbb{C}^{\mathbb{C}} \arrow[rr] & & \mathbb{C}^{\mathbb{C}} \\
H \arrow[dd, "\alpha"'] & & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} HP \arrow[dd, "Id \dot{+} Id \dot{\times} \alpha P"] \\
& \mapsto & \\
K & & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} KP
\end{tikzcd}$$

Another way of writing this is by lifting the sum and product functors *two* levels rather than one, so

$$N = \underline{\underline{\mathbb{1}}} \ddot{+} \underline{\underline{Id_{\mathbb{C}}}} \ddot{\times} (\circ P) \tag{3.4.1}$$

where $(\circ P)$ is the functor that precomposes P:

$$\begin{tikzcd}
(\circ P) : \mathbb{C}^{\mathbb{C}} \arrow[rr] & & \mathbb{C}^{\mathbb{C}} \\
F \arrow[dd, "\alpha"'] & & FP \arrow[dd, "\alpha P"] \\
& \mapsto & \\
G & & GP
\end{tikzcd}$$

To see that the more compact definition of N using double lifting is the same as our original definition, we can calculate:

$$\Big(\underline{\underline{\mathbb{1}}} \ddot{+} \underline{\underline{Id_{\mathbb{C}}}} \ddot{\times} (\circ P)\Big).F$$

$=$ { lifted sums and products }

$$\underline{\underline{\mathbb{1}}}.F \dot{+} \underline{\underline{Id_{\mathbb{C}}}}.F \dot{\times} (\circ P).F$$

$=$ { constant functors }

$$\begin{array}{ll} & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} (\circ P).F \\ = & \qquad \{\text{ definition of } (\circ P) \,\} \\ & \underline{\mathbb{1}} \dot{+} Id_{\mathbb{C}} \dot{\times} FP \end{array}$$

for any object $F : \mathbb{C} \to \mathbb{C}$ in $\mathbb{C}^{\mathbb{C}}$. A similar calculation works for arrows in $\mathbb{C}^{\mathbb{C}}$. What equation 3.4.1 shows us is that the base functor for nests is built out of constant higher-order functors ($\underline{\underline{\mathbb{1}}}$ and $\underline{Id_{\mathbb{C}}}$), sums and products of higher-order functors, and the higher-order functor $(\circ P)$. If we can show that all of these elements are ω-cocontinuous then N must be ω-cocontinuous and we can appeal to corollary 3.4 to prove the existence of an initial algebra for N.

Constant functors are trivially ω-cocontinuous. Sum functors are left adjoints to diagonal functors and are therefore cocontinuous, in particular, ω-cocontinuous. Product functors can be shown to be ω-cocontinuous in ω-cocomplete cartesian closed categories [ManesA86]. Lifting functors preserves their continuity properties:

> **Lemma 3.3 (lifting preserves cocontinuity):** If $\mathbb{C}$ is $\mathbb{J}$-cocomplete and $F : \mathbb{C} \to \mathbb{D}$ is $\mathbb{J}$-cocontinuous for some small category $\mathbb{J}$, then so is the lifting of F, $\dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$, for any small category $\mathbb{B}$.
>
> **Proof:** In appendix A.2.

There is a dual result for $\mathbb{J}$-continuity. So lifted or doubly-lifted sums and products are ω-cocontinuous if sums and products are.

What differentiates the structure of base functors for non-uniform types from those for uniform types is the appearance of precomposition functors such as $(\circ P)$. These are the parts of the base functor that cause the non-uniform recursion. We need to show that these precomposition functors are ω-cocontinuous. In general, for a non-uniform type with m modifying functors, $M_1, \ldots, M_m$, we will need to prove the ω-cocontinuity of $(\circ M_1), \ldots, (\circ M_m)$. The modifying functors can be arbitrary functors, and it is difficult to give general conditions for $(\circ M)$ to be ω-cocontinuous for some modifying functor M. We do have the following results in the case of adjoint functors [MacLane71] though. Adjoint pairs of functors lift to adjoint pairs of higher-order functors in two different ways — using postcomposition (what we call *functor lifting*) and precomposition:

> **Theorem 3.3 ([Schubert72, proposition 16.5.11]):** Let $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{D} \to \mathbb{C}$ be adjoint functors so that $F \dashv G$. Then
>
> (i) for any small category $\mathbb{B}$, the lifted functors
>
> $$\dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}} \qquad\qquad \dot{G} : \mathbb{D}^{\mathbb{B}} \to \mathbb{C}^{\mathbb{B}}$$
>
> are left and right adjoints so that $\dot{F} \dashv \dot{G}$
>
> (ii) if $\mathbb{C}$ and $\mathbb{D}$ are small then for any category $\mathbb{E}$, the functors
>
> $$(\circ G) : \mathbb{E}^{\mathbb{C}} \to \mathbb{E}^{\mathbb{D}} \qquad\qquad (\circ F) : \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{C}}$$
>
> are left and right adjoints so that $(\circ G) \dashv (\circ F)$
>
> Note that in part (ii), the order of the adjoints is reversed.

We are interested in finding out when a functor $(\circ M)$ is ω-cocontinuous. Because left adjoints are always cocontinuous, the above theorem tells us:

> **Corollary 3.5:** If $M : \mathbb{C} \to \mathbb{D}$ is right adjoint to some functor $L : \mathbb{D} \to \mathbb{C}$, then $(\circ M)$ is ω-cocontinuous.
>
> **Proof:** If $L \dashv M$ then by theorem 3.3 we know that $(\circ M) \dashv (\circ L)$. So $(\circ M)$ is a left adjoint and is therefore cocontinuous, in particular, ω-cocontinuous.

However, this result will not be applicable in many cases because few modifying functors will be right adjoints. We will see, in the following section, that it is easier to prove $(\circ M)$ to be *locally* ω-continuous in the setting of order-enriched functor categories, and appeal to the order-enriched results for the existence of initial algebras.

We summarize the results we have about ω-cocontinuity of higher-order functors:

Lemma 3.4: A higher-order functor K is ω-cocontinuous if it satisfies any of the following:

(i) K is an identity functor;

(ii) K is a constant functor;

(iii) K is a composite functor ML where both M and L are ω-cocontinuous;

(iv) K is a lifted functor $\dot{F}$ for some ω-cocontinuous functor $F : \mathbb{C} \to \mathbb{D}$ (including bifunctors) where $\mathbb{C}$ is ω-cocomplete;

(v) K is a precomposition functor $(\circ M)$ for some functor M that is a right adjoint.

Proof: Parts (i) – (iii) are standard. Parts (iv) and (v) are proved by lemma 3.3 and corollary 3.5 respectively.

We now look at the problem of showing that type functors of non-uniform data-types are ω-cocontinuous. Suppose that we can show that the base functor $F : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}^{\mathbb{C}}$ for our non-uniform type is ω-cocontinuous, and that $\mathbb{D}$ is ω-cocomplete and has an initial object $\mathbb{0}$, then the initial algebra of F is computed as the colimit of the ω-diagram:

$$\underline{\mathbb{0}} \xrightarrow{\dot{!}_{F.\underline{\mathbb{0}}}} F.\underline{\mathbb{0}} \xrightarrow{F.\dot{!}_{F.\underline{\mathbb{0}}}} FF.\underline{\mathbb{0}} \xrightarrow{FF.\dot{!}_{F.\underline{\mathbb{0}}}} \cdots$$

and the carrier of the initial algebra, μF, is the *type functor* of the non-uniform data-type. If we wish to use μF to construct further recursive data-types, such as lists or trees of μF-structures, then we must show that μF is ω-cocontinuous so that we can apply the standard theory. Our intuition tells us that if μF is the colimit of a sequence of ω-cocontinuous functors,

$$\underline{\mathbb{0}}, F.\underline{\mathbb{0}}, FF.\underline{\mathbb{0}}, \ldots$$

then μF should also be ω-cocontinuous. This idea is formalized in the following theorem:

> **Theorem 3.4:** Let $\mathbb{B}$ be a small category and $\mathbb{C}$ a $\mathbb{J}$-cocomplete category for some small category $\mathbb{J}$. Define $\mathbb{J}$-*cocont*$(\mathbb{B}, \mathbb{C})$ to be the full subcategory of $\mathbb{C}^{\mathbb{B}}$ whose objects are the $\mathbb{J}$-cocontinuous functors from $\mathbb{B}$ to $\mathbb{C}$. Then $\mathbb{J}$-*cocont*$(\mathbb{B}, \mathbb{C})$ is $\mathbb{J}$-cocomplete, with $\mathbb{J}$-colimits being formed in the same way as in $\mathbb{C}^{\mathbb{B}}$ (that is, pointwise). In particular, $\mathbb{J}$-colimits are preserved by the inclusion $\mathbb{J}$-*cocont*$(\mathbb{B}, \mathbb{C}) \subseteq \mathbb{C}^{\mathbb{B}}$.
>
> **Proof:** See Schubert's section on *double limits* and *colimits* [Schubert72, sections 7.6 and 8.6].

The essence of this result is that $\mathbb{J}$-diagrams of $\mathbb{J}$-cocontinuous functors in $\mathbb{C}^{\mathbb{B}}$ have colimiting objects that are also $\mathbb{J}$-cocontinuous functors:

> **Corollary 3.6:** Let $\mathbb{B}$ be small and $\mathbb{C}$ a $\mathbb{J}$-cocomplete category for some small $\mathbb{J}$. Let $D : \mathbb{J} \to \mathbb{C}^{\mathbb{B}}$ be any $\mathbb{J}$-diagram in $\mathbb{C}^{\mathbb{B}}$ such that for each object j in $\mathbb{J}$, the functor $D.j : \mathbb{B} \to \mathbb{C}$ is $\mathbb{J}$-cocontinuous, then the colimiting object of the diagram D, $Colim(D) : \mathbb{B} \to \mathbb{C}$, is also $\mathbb{J}$-cocontinuous.

So if we can show that $\underline{\mathbb{0}}, F.\underline{\mathbb{0}}, FF.\underline{\mathbb{0}}$, and so on, are all ω-cocontinuous then it follows that μF must be ω-cocontinuous. The constant functor $\underline{\mathbb{0}}$ is trivially ω-cocontinuous, so all we need to prove is that the higher-order functor F preserves ω-cocontinuity of functors. Then we have

> **Lemma 3.5:** If $\mathbb{D}$ is ω-cocomplete and has an initial object $\mathbb{0}$, and $F : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}^{\mathbb{C}}$ is ω-cocontinuous and *preserves* ω-cocontinuity of functors then F has an initial algebra and the carrier, $\mu F : \mathbb{C} \to \mathbb{D}$, is it-

self ω-cocontinuous.

Proof: Apply theorem 3.4 to get the initial algebra from the colimit of the diagram:

$$\begin{tikzcd}\underline{\mathbb{0}} \arrow[r, "!_{F.\underline{\mathbb{0}}}"] & F.\underline{\mathbb{0}} \arrow[r, "F.!_{F.\underline{\mathbb{0}}}"] & FF.\underline{\mathbb{0}} \arrow[r, "FF.!_{F.\underline{\mathbb{0}}}"] & \cdots\end{tikzcd}$$

We know $\underline{\mathbb{0}}$ is ω-cocontinuous, and that F preserves ω-cocontinuity, so for each i in ω, $F^i.\underline{\mathbb{0}}$ is ω-cocontinuous. Then apply corollary 3.6 to see that μF must be ω-cocontinuous.

For any ω-cocontinuous functor M, the precomposition functor $(\circ M)$ preserves ω-cocontinuity:

Lemma 3.6: If $M : \mathbb{B} \to \mathbb{C}$ is ω-cocontinuous then $(\circ M) : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}^{\mathbb{B}}$ preserves ω-cocontinuity.

Proof: Let $F : \mathbb{C} \to \mathbb{D}$ be ω-cocontinuous, then $(\circ M).F = FM$ which is ω-cocontinuous because the composition of two ω-cocontinuous functors is ω-cocontinuous.

Similarly for *lifted* ω-cocontinuous functors:

Lemma 3.7: If $F : \mathbb{C} \to \mathbb{D}$ is ω-cocontinuous then $\dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$ preserves ω-cocontinuity.

Proof: Let $G : \mathbb{B} \to \mathbb{C}$ be ω-cocontinuous, then $\dot{F}.G = FG$ which is ω-cocontinuous because the composition of two ω-cocontinuous functors is ω-cocontinuous.

For bifunctors such as sums and products we have:

Lemma 3.8: If $\dot{\otimes} : \mathbb{E}^{\mathbb{D}} \times \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{D}}$ preserves ω-cocontinuity and the higher-order functors $H, K : \mathbb{C}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{D}}$ preserve ω-continuity then so does $H \ddot{\otimes} K : \mathbb{C}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{D}}$.

Proof: Let $F : \mathbb{B} \to \mathbb{C}$ be ω-cocontinuous, then

$$(H \ddot{\otimes} K).F = (H.F) \dot{\otimes} (K.F)$$

Both H and K preserve ω-cocontinuity so $H.F$ and $K.F$ are ω-cocontinuous. Then $(H.F) \dot{\otimes} (K.F)$ is also ω-cocontinuous.

Lifted sums always preserve ω-cocontinuity [ManesA86, theorem 11.2.7]. Lifted products $\dot{\times} : \mathbb{D}^{\mathbb{C}} \times \mathbb{D}^{\mathbb{C}} \to \mathbb{D}^{\mathbb{C}}$ preserve ω-cocontinuity when $\mathbb{D}$ is any ω-cocomplete Cartesian-closed category, for example **Set** or ω-**CPPO** [ManesA86, theorem 11.2.10 and section 13.3.9]. Lastly we need to show that the composition of two ω-cocontinuity preserving higher-order functors also preserves ω-cocontinuity:

Lemma 3.9: If $H : \mathbb{C}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{D}}$ and $K : \mathbb{E}^{\mathbb{D}} \to \mathbb{G}^{\mathbb{F}}$ both preserve ω-cocontinuity then so does $KH : \mathbb{C}^{\mathbb{B}} \to \mathbb{G}^{\mathbb{F}}$.

Proof: Let $L : \mathbb{B} \to \mathbb{C}$ be an ω-cocontinuous functor, then so is $(H.L) : \mathbb{D} \to \mathbb{E}$ and therefore also $K.(H.L) : \mathbb{F} \to \mathbb{G}$.

Thus we have proved that all polynomial higher-order functors are ω-cocontinuity preserving, and this remains true if we allow precomposition and lifted functors as well. Therefore any non-uniform type that arises as the initial algebra of such a base functor has a type functor that is ω-cocontinuous.

3.4.2. Order-enriching functor categories

We have seen that it may be difficult, in general, to show that the higher-order functor $(\circ M)$, for some modifying functor M, is ω-cocontinuous. In this section we take an alternative approach by lifting the order-enriched results from section 2.5.2 to functor categories, and then proving our base functors to be *locally* ω-continuous.

We need a version of theorem 2.4 for functor categories. As we did for the functor category version of the generalized Kleene fixed-point theorem, we will try to rephrase the properties required of the functor category in terms of properties of the target category.

First we show that functor categories inherit a pointwise ordering from their target category:

Lemma 3.10 (a functor category is order-enriched if its target category is): If $\mathbb{D}$ is an **O**-category and $\mathbb{C}$ is a small category, then $\mathbb{D}^{\mathbb{C}}$ is also an **O**-category — the ordering on the hom-sets $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$ of $\mathbb{D}^{\mathbb{C}}$ being defined componentwise from the ordering on the hom-sets of $\mathbb{D}$:

$$\alpha \sqsubseteq_{\mathbb{D}^{\mathbb{C}}} \beta \;\equiv\; \forall A \in \mathbb{C}, \alpha_A \sqsubseteq_{\mathbb{D}} \beta_A$$

for any objects F and G in $\mathbb{D}^{\mathbb{C}}$. Furthermore, least upper bounds of ascending ω-chains in the hom-sets of $\mathbb{D}^{\mathbb{C}}$ can be computed pointwise:

$$\left(\bigsqcup \left\{ \alpha^i : F \overset{\cdot}{\to} G \right\}_{i \in \omega} \right)_A = \bigsqcup \left\{ \alpha^i_A : F.A \to G.A \right\}_{i \in \omega} \tag{3.4.2}$$

Proof: In appendix A.4.

This means that "evaluate at A" functors are *locally ω-continuous*:

Corollary 3.7: Let A be any object in $\mathbb{C}$, then the "evaluate at A" functor, $(@A) : \mathbb{D}^{\mathbb{C}} \to \mathbb{D}$ is locally ω-continuous.

Proof: We must show that $(@A)$ preserves least upper bounds of ω-chains. Let $\left\{ \alpha^i : F \overset{\cdot}{\to} G \right\}_{i \in \omega}$ be any ω-chain. Then

$$(@A).\left(\bigsqcup \left\{ \alpha^i \right\}_{i \in \omega} \right)$$

$=$ { definition of $(@A)$ }

$$\left(\bigsqcup \left\{ \alpha^i \right\}_{i \in \omega} \right)_A$$

$=$ { equation 3.4.2 }

$$\bigsqcup\left\{ \alpha^i_A \right\}_{i\in\omega}$$

$=$ { definition of $(@A)$ }

$$\bigsqcup\left\{ (@A).\alpha^i \right\}_{i\in\omega}$$

In particular, each $(@A)$ is *locally monotonic*, and therefore preserves embeddings:

Lemma 3.11 ("evaluate at A" functors preserve embeddings): Let $\mathbb{D}$ be an **O**-category and $\mathbb{C}$ any small category. We know that $\mathbb{D}^{\mathbb{C}}$ is also an **O**-category with the componentwise ordering. If $\alpha : F \xrightarrow{\bullet} G$ is an embedding in $\mathbb{D}^{\mathbb{C}}$, then for any object A in $\mathbb{C}$, $\alpha_A : F.A \to G.A$ is also an embedding. Note that the retraction of α_A is

$$\left(\alpha_A\right)^R = \left(\alpha^R\right)_A \tag{3.4.3}$$

So we can unambiguously write α^R_A without brackets.

Proof: It is easy to show that each $(@A)$ is locally monotonic. Because locally monotonic functors preserve projection pairs (lemma 2.10), we know that if $\left(\alpha, \alpha^R\right)$ is a projection pair then so is $\left((@A).\alpha, (@A).\alpha^R\right)$, that is,

$$\alpha_A : F.A \to G.A$$

is an embedding with corresponding retraction

$$\left(\alpha^R\right)_A : G.A \to F.A$$

Furthermore,

$$\left(\alpha^R\right)_A$$

$=$ { definition of $(@A)$ }

$$(@A).\alpha^R$$

$=$ { because retraction of $(@A).\alpha$ is $(@A).\alpha^R$ }

$((@A).\alpha)^R$

$=$ { definition of $(@A)$ }

$(\alpha_A)^R$

Lemma 3.12 (natural transformations form projection pairs if, and only if, all their components do): Let $\mathcal{C}$ be an **O**-category and $\mathcal{B}$ a small category. Let H and K both be objects of $\mathcal{C}^{\mathcal{B}}$, that is, functors from $\mathcal{B}$ to $\mathcal{C}$, and $\alpha : H \xrightarrow{\bullet} K$ and $\beta : K \xrightarrow{\bullet} H$ natural transformations between them. Then (α, β) is a projection pair from H to K in $\mathcal{C}^{\mathcal{B}}$ if, and only if, for every object A in $\mathcal{B}$, $(\alpha_A : H.A \to K.A, \beta_A : K.A \to H.A)$ is a projection pair from $H.A$ to $K.A$ in $\mathcal{C}$.

Proof: We prove the first property of projection:

$\beta \circ \alpha = Id_H$

$\equiv$ { equality of natural transformations }

$\forall A \in \mathcal{B}, (\beta \circ \alpha)_A = (Id_H)_A$

$\equiv$ { identities and composition of natural transformations }

$\forall A \in \mathcal{B}, \beta_A \circ \alpha_A = Id_{H.A}$

and the second:

$\alpha \circ \beta \sqsubseteq Id_K$

$\equiv$ { ordering on hom-sets of $\mathcal{C}^{\mathcal{B}}$ }

$\forall A \in \mathcal{B}, (\alpha \circ \beta)_A \sqsubseteq (Id_K)_A$

$\equiv$ { identities and composition of natural transformations }

$\forall A \in \mathcal{B}, \alpha_A \circ \beta_A \sqsubseteq Id_{K.A}$

and that completes the proof.

We would like to show that if $\mathbb{D}$ is an **O**$\perp$-category then so is $\mathbb{D}^{\mathbb{C}}$ for any small $\mathbb{C}$. Unfortunately, naturality requirements force us to restrict our attention to the strict subcategory of $\mathbb{D}$:

Lemma 3.13: If $\mathbb{D}$ is an **O**$\perp$-category then for any small category $\mathbb{C}$, the functor category $(\mathbb{D}_{\perp})^{\mathbb{C}}$ is also an **O**$\perp$-category. For any functors F and G, the least element of the hom-set $hom_{(\mathbb{D}_{\perp})^{\mathbb{C}}}(F, G)$ has components:

$$\left(\perp_{F \to G}\right)_A \overset{def}{=} \perp_{F.A \to G.A}$$

for each A in $\mathbb{C}$.

Proof: In appendix A.4.

In particular, this means that every arrow in $(\mathbb{D}_{\perp})^{\mathbb{C}}$ is strict:

Lemma 3.14: Every arrow in the category $(\mathbb{D}_{\perp})^{\mathbb{C}}$ described above is strict.

Proof: Let $\alpha : F \to G$ be an arrow in $(\mathbb{D}_{\perp})^{\mathbb{C}}$. Then each component $\alpha_A : F.A \to G.A$ of α is an arrow in $\mathbb{D}_{\perp}$ and must therefore be strict. To show α to be strict, we show that it preserves least elements $\perp_{H \to F}$ for any H in $(\mathbb{D}_{\perp})^{\mathbb{C}}$:

$$\alpha \circ \perp_{H \to F} = \perp_{H \to G}$$

$\equiv$ { componentwise equality of natural transformations }

$$\forall A \in \mathbb{C}, \left(\alpha \circ \perp_{H \to F}\right)_A = \left(\perp_{H \to G}\right)_A$$

$\equiv$ { composition of natural transformations, definition of $\perp$ }

$$\forall A \in \mathbb{C}, \alpha_A \circ \perp_{H.A \to F.A} = \perp_{H.A \to G.A}$$

$\equiv$ { each α_A is strict in $\mathbb{D}_{\perp}$ }

True

So α is strict.

Then, because every arrow must be strict, they must all appear in the strict subcategory of $(\mathbb{D}_\perp)^\mathbb{C}$:

Corollary 3.8: The *strict subcategory* of $(\mathbb{D}_\perp)^\mathbb{C}$ is $(\mathbb{D}_\perp)^\mathbb{C}$ itself.

We can also show that functor categories are *localized* or *have localized ω-colimits of embeddings* if their target categories are:

Lemma 3.15 (functor categories are localized if their target categories are): Let $\mathbb{D}$ be a localized **O**-category and $\mathbb{C}$ any small category. Then $\mathbb{D}^\mathbb{C}$ is a localized category.

Proof: See appendix A.4.

We need conditions such that $\left(\left(\mathbb{D}_\perp\right)^\mathbb{C}\right)_E$ has an initial object. We have the following:

Lemma 3.16: Let $\mathbb{D}$ be an **O**$\perp$-category and $\mathbb{C}$ any small category. If $\mathbb{D}$ or $\mathbb{D}_\perp$ has a terminal object $\mathbb{1}$ then the constant functor $\underline{\mathbb{1}}$ is an initial object in $\left(\left(\mathbb{D}_\perp\right)^\mathbb{C}\right)_E$

Proof: If $\mathbb{D}$ has a terminal object $\mathbb{1}$ then lemma 2.20 tells us that it is also terminal in $\mathbb{D}_\perp$. Then $\underline{\mathbb{1}}$ is terminal in $\left(\mathbb{D}_\perp\right)^\mathbb{C}$ because limits in functor categories are computed pointwise. Lastly, theorem 2.3 tells us that $\underline{\mathbb{1}}$ is initial in $\left(\left(\mathbb{D}_\perp\right)^\mathbb{C}\right)_E$

We will also need $\left(\left(\mathbb{D}_\perp\right)^\mathbb{C}\right)_E$ to be ω-cocomplete. Again, we can ensure this by completeness conditions on $\mathbb{D}$, in fact $\mathbb{D}_\perp$:

Lemma 3.17: If $\mathbb{D}$ is an **O**$\perp$-category such that $\mathbb{D}_\perp$is ω-cocomplete or ω^{op}-complete then $\left(\left(\mathbb{D}_\perp\right)^\mathbb{C}\right)_E$ is ω-cocomplete for any small category $\mathbb{C}$.

Proof: Assume $\mathbb{D}_\perp$ is ω-cocomplete or ω^{op}-complete, then so is $\left(\mathbb{D}_\perp\right)^\mathbb{C}$ because functor categories inherit completeness properties from their target categories. Then $\left(\left(\mathbb{D}_\perp\right)^{\mathbb{C}}\right)_E$ is ω-cocomplete by lemma 2.7.

Now we have enough results to state a functor category version of theorems 2.4 and 2.5:

Theorem 3.5: Let $\mathbb{C}$ be a small category and $\mathbb{D}$ a localized **O**$\perp$-category such that

(i) $\mathbb{D}$ or $\mathbb{D}_\perp$ has a terminal object $\mathbb{1}$;

(ii) $\mathbb{D}_\perp$ is ω-cocomplete or ω^{op}-complete;

and let $F : \left(\mathbb{D}_\perp\right)^{\mathbb{C}} \to \left(\mathbb{D}_\perp\right)^{\mathbb{C}}$ be a locally ω-cocontinuous endofunctor. Then F has an initial algebra. Furthermore, for any F-algebra $\alpha : F.H \xrightarrow{\bullet} H$ in $\left(\mathbb{D}_\perp\right)^{\mathbb{C}}$, the following statements are equivalent:

(i) α is an initial F-algebra;

(ii) α is an isomorphism and the least fixed-point of the mapping

$$\beta \mapsto \alpha \circ F.\beta \circ \alpha^{-1}$$

is the identity on H.

Proof: Because $\mathbb{D}$ is a localized **O**$\perp$-category, so is $\mathbb{D}_\perp$ by lemmas 2.6 and 2.19. Then by lemmas 3.13 and 3.15 we know that $\left(\mathbb{D}_\perp\right)^{\mathbb{C}}$ is also a localized **O**$\perp$-category. Lemmas 3.17 and 3.16 show that $\left(\left(\mathbb{D}_\perp\right)^{\mathbb{C}}\right)_E$ is ω-cocomplete and has an initial object. Corollary 3.8 tells us that $\left(\left(\mathbb{D}_\perp\right)^{\mathbb{C}}\right)_\perp = \left(\mathbb{D}_\perp\right)^{\mathbb{C}}$ and so F can be considered as a functor

$$F : \left(\left(\mathbb{D}_\perp\right)^{\mathbb{C}}\right)_\perp \to \left(\left(\mathbb{D}_\perp\right)^{\mathbb{C}}\right)_\perp$$

We have now shown that $\left(\mathbb{D}_\perp\right)^{\mathbb{C}}$ satisfies the conditions of theorems 2.4 and 2.5, and so F has an initial algebra and the above statements are equivalent.

3.4.2.1. *Polymorphic folds in Haskell are higher-order catamorphisms*

Recall the fold combinator we defined for nests:

```
foldNest :: (∀ a . r a) →
                (∀ a . a → r (P a) → r a) →
                   Nest b → r b
foldNest lam rho NilN = lam
foldNest lam rho (ConsN x xs) = rho x (foldNest lam rho xs)
```

By definition this gives an algebra homomorphism from the N-algebra NilN $\dot{\triangledown}$ ConsN to any other (strict) N-algebra. To show that folds produced using this combinator are actually catamorphisms, we must show that NilN $\dot{\triangledown}$ ConsN is an initial N-algebra. Provided that the base functor for nests is locally ω-continuous, we can appeal to theorem 3.5. Then it is sufficient to show that NilN $\dot{\triangledown}$ ConsN is an isomorphism and

```
(foldNest NilN ConsN) :: Nest a → Nest a
```

is the identity. As before, pattern-matching provides the inverse to NilN $\dot{\triangledown}$ ConsN. To show that (foldNest NilN ConsN) is the identity on finite nests is straightforward, and can be proved pointwise for all instances by induction on the “length” of the nest. For infinite lists we again look at their ascending ω-chain of finite approximations, for which they are the limit. Then (foldNest NilN ConsN) is the identity on the finite approximations and so is also the identity on the limit.

A similar argument holds for polymorphic folds defined in this way over other non-uniform types.

3.4.2.2. Locally ω-continuous higher-order functors

In order to be able to use theorem 3.5, we must show that our base functors are locally ω-continuous. Identity and constant higher-order functors are trivially locally ω-continuous. We can also prove that lifting preserves local ω-cocontinuity. First we prove that it preserves local monotonicity:

> **Lemma 3.18 (lifting preserves local monotonicity):** Let $\mathbb{C}$ and $\mathbb{D}$ be **O**-categories and $\mathbb{B}$ a small category. Then for any locally monotonic functor $F : \mathbb{C} \to \mathbb{D}$, the lifted functor $\dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$ is locally monotonic.
>
> **Proof:** Let $\alpha, \beta : H \xrightarrow{\bullet} K$ be two arrows in the hom-set $hom_{\mathbb{C}^{\mathbb{B}}}(H, K)$ for some H and K in $\mathbb{C}^{\mathbb{B}}$, such that $\alpha \sqsubseteq_{\mathbb{C}^{\mathbb{B}}} \beta$. Then we wish to show that $\dot{F}.\alpha \sqsubseteq_{\mathbb{D}^{\mathbb{B}}} \dot{F}.\beta$. We do this with a pointwise calculation:
>
> $$\begin{array}{ll} & (\dot{F}.\alpha)_A \\ = & \quad \{ \text{definition of } \dot{F} \} \\ & (F\alpha)_A \\ = & \quad \{ \text{composition of functors and natural transformations} \} \\ & F.\alpha_A \\ \sqsubseteq_{\mathbb{D}} & \quad \{ \alpha_A \sqsubseteq_{\mathbb{C}} \beta_A \text{ and } F \text{ is locally monotonic} \} \\ & F.\beta_A \\ = & \quad \{ \text{composition of functors and natural transformations} \} \\ & (F\beta)_A \\ = & \quad \{ \text{definition of } \dot{F} \} \\ & (\dot{F}.\beta)_A \end{array}$$
>
> for every A in $\mathbb{B}$. Then, because the ordering on $\mathbb{D}^{\mathbb{C}}$ is pointwise, we are finished.

Then we can prove that lifting preserves local ω-continuity:

Lemma 3.19 (lifting preserves local ω-continuity): Let $\mathbb{C}$ and $\mathbb{D}$ be **O**-categories, and $F : \mathbb{C} \to \mathbb{D}$ a locally ω-continuous functor. Then for any small category $\mathbb{B}$, the lifting of F,

$$\dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$$

is also locally ω-continuous.

Proof: Let H and K be any two objects of $\mathbb{C}^{\mathbb{B}}$, that is, functors from $\mathbb{B}$ to $\mathbb{C}$. Then the hom-set $hom_{\mathbb{C}^{\mathbb{B}}}(H, K)$ is the set of all natural transformations from H to K. We must show that $\dot{F}$ restricted to such a hom-set is ω-continuous.

Let $\alpha = \left\{ \alpha^i : H \xrightarrow{\bullet} K \right\}_{i \in \omega}$ be an ω-chain in $hom_{\mathbb{C}^{\mathbb{B}}}(H, K)$. Then α has a least upper bound because $\mathbb{C}^{\mathbb{B}}$ is an **O**-category. We must show that $\dot{F}$ preserves that least upper bound, that is,

$$\dot{F}.\left(\sqcup \left\{ \alpha^i \right\}_{i \in \omega} \right) = \sqcup \left\{ \dot{F}.\alpha^i \right\}_{i \in \omega} \tag{3.4.4}$$

First we must justify that $\left\{ \dot{F}.\alpha^i \right\}_{i \in \omega}$ is an ascending ω-chain. We know F is locally monotonic and lemma 3.18 tells us that $\dot{F}$ is also locally monotonic. Then locally monotonic functors preserve ascending ω-chains.

The left and right sides of equation 3.4.4 are natural transformations of type $\dot{F}.H \xrightarrow{\bullet} \dot{F}.K$, and we prove them equal by proving their components equal. For each A in $\mathbb{B}$,

$$\left(\dot{F}.\left(\sqcup \left\{ \alpha^i \right\}_{i \in \omega} \right) \right)_A$$

$=$ { definition of $\dot{F}$ }

$$\left(F\left(\sqcup \left\{ \alpha^i \right\}_{i \in \omega} \right) \right)_A$$

$=$ { composition of functors and natural transformations }

$$F.\left(\sqcup \left\{ \alpha^i \right\}_{i \in \omega} \right)_A$$

$=$ { least upper bounds in functor categories }

$$F.\bigsqcup\left\{\alpha^i_A\right\}_{i\in\omega}$$

$=$ { F is locally ω-continuous }

$$\bigsqcup\left\{F.\alpha^i_A\right\}_{i\in\omega}$$

$=$ { composition of functors and natural transformations }

$$\bigsqcup\left\{\left(F\alpha^i\right)_A\right\}_{i\in\omega}$$

$=$ { definition of $\dot{F}$ }

$$\bigsqcup\left\{\left(\dot{F}.\alpha^i\right)_A\right\}_{i\in\omega}$$

$=$ { least upper bounds in functor categories }

$$\left(\bigsqcup\left\{\dot{F}.\alpha^i\right\}_{i\in\omega}\right)_A$$

The same is true for any lifted *bifunctor*.

The standard bifunctors used to construct data-types in ω-**CPPO** and ω-**CPPO**$_\perp$, namely Cartesian product, smash product, coalesced sum and separated sum are all locally ω-continuous (see [SmythP82, example 2] and [FokkingaM91]) and so, therefore are their liftings to functor categories.

The following result about doubly-lifted bifunctors is also useful:

Lemma 3.20: Let $\mathbb{C}$ and $\mathbb{E}$ be **O**-categories and $\mathbb{B}$ and $\mathbb{D}$ be small categories. If $H, K : \mathbb{C}^{\mathbb{B}} \to \mathbb{E}^{\mathbb{D}}$ and $\dot{\otimes} : \mathbb{E}^{\mathbb{D}} \times \mathbb{E}^{\mathbb{D}} \to \mathbb{E}^{\mathbb{D}}$ are all locally ω-continuous then

$$\begin{tikzcd}
H \ddot{\otimes} K : \mathbb{C}^{\mathbb{B}} \arrow[r] & \mathbb{E}^{\mathbb{D}} \\
F \arrow[d, "\alpha"'] & (H.F)\dot{\otimes}(K.F) \arrow[d, "(H.\alpha)\dot{\otimes}(K.\alpha)"] \\
G & (H.G)\dot{\otimes}(K.G)
\end{tikzcd}$$

with $\mapsto$ between the two arrows.

is also locally ω-continuous.

Proof: The same as lemma 2.9, but at the level of functor categories.

We can also show that the act of precomposing *any* modifying functor M is locally ω-continuous. Again, we begin by showing that $(\circ M)$ is locally monotonic:

> **Lemma 3.21 ($(\circ M)$ functors are locally monotonic):** Let $\mathbb{D}$ be an **O**-category and $M : \mathbb{B} \to \mathbb{C}$ be a functor between two small categories. Then the functor
>
> $$(\circ M) : \mathbb{D}^{\mathbb{C}} \longrightarrow \mathbb{D}^{\mathbb{B}}$$
>
> $$\begin{tikzcd} F \arrow[d, "\alpha"'] & & FM \arrow[d, "\alpha M"] \\ G \arrow[rr, phantom, "\mapsto" near start, shift left=4ex] & & GM \end{tikzcd}$$
>
> is locally monotonic.
>
> **Proof:** Let $\alpha, \beta : F \xrightarrow{\bullet} G$ be arrows in $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$ such that $\alpha \sqsubseteq_{\mathbb{D}^{\mathbb{C}}} \beta$. Then
>
> $(\circ M).\alpha \sqsubseteq_{\mathbb{D}^{\mathbb{B}}} (\circ M).\beta$
>
> $\equiv$ { definition of $(\circ M)$ }
>
> $\alpha M \sqsubseteq_{\mathbb{D}^{\mathbb{B}}} \beta M$
>
> $\equiv$ { pointwise ordering }
>
> $\forall A \in \mathbb{B}, \alpha M_A \sqsubseteq_{\mathbb{D}} \beta M_A$
>
> $\equiv$ { composition of functors and natural transformations }
>
> $\forall A \in \mathbb{B}, \alpha_{M.A} \sqsubseteq_{\mathbb{D}} \beta_{M.A}$
>
> $\Leftarrow$ { each $M.A$ is an object in $\mathbb{C}$ }
>
> $\forall B \in \mathbb{C}, \alpha_B \sqsubseteq_{\mathbb{D}} \beta_B$
>
> $\equiv$ { pointwise ordering }
>
> $\alpha_B \sqsubseteq_{\mathbb{D}^{\mathbb{C}}} \beta_B$
>
> $\equiv$ { assumption }
>
> True
>
> So $(\circ M)$ is locally monotonic.

and then local ω-continuity follows:

Theorem 3.6 ($(\circ M)$ functors are locally ω-continuous): Let $\mathbb{D}$ be an **O**-category and $M : \mathbb{B} \to \mathbb{C}$ be a functor between two small categories. Then the functor

$$\begin{tikzcd}
(\circ M) : \mathbb{D}^{\mathbb{C}} \arrow[r] & \mathbb{D}^{\mathbb{B}} \\
F \arrow[d, "\alpha"'] \arrow[r, phantom, "\mapsto" description, shift right=4ex] & FM \arrow[d, "\alpha M"] \\
G & GM
\end{tikzcd}$$

is locally ω-continuous.

Proof: Because $\mathbb{D}$ is an **O**-category, we know that both $\mathbb{D}^{\mathbb{B}}$ and $\mathbb{D}^{\mathbb{C}}$ are **O**-categories. We must show that $(\circ M)$ preserves least upper bounds of ω-chains in the hom-sets of $\mathbb{D}^{\mathbb{C}}$.

Let $F, G : \mathbb{C} \to \mathbb{D}$ be two functors, and

$$\left\{ \alpha^i : F \xrightarrow{\bullet} G \right\}_{i \in \omega}$$

an ascending ω-chain in the hom-set $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$. We know that this ω-chain has a least upper bound because $\mathbb{D}^{\mathbb{C}}$ is an **O**-category. We must show that

$$(\circ M).\bigsqcup\left\{ \alpha^i \right\}_{i\in\omega} = \bigsqcup\left\{ (\circ M).\alpha^i \right\}_{i\in\omega}$$

or, expanding the definition of $(\circ M)$,

$$\left(\bigsqcup\left\{ \alpha^i \right\}_{i\in\omega} \right) M = \bigsqcup\left\{ \alpha^i M \right\}_{i\in\omega}$$

First we check that the least upper bound on the right hand side exists. We know that $\left\{ \alpha^i \right\}_{i\in\omega}$ is an ascending ω-chain. We also know, by corollary 3.21, that $(\circ M)$ is locally monotonic, so $\left\{ (\circ M).\alpha^i \right\}_{i\in\omega}$ is also an ascending ω-chain. But this is the ω-chain $\left\{ \alpha^i M \right\}_{i\in\omega}$. Then, because $\mathbb{D}^{\mathbb{B}}$ is an **O**-category, this ω-chain has a least upper bound, and the right hand side exists.

Now we can perform a pointwise calculation:

$$\begin{aligned}
& \left(\left(\sqcup\{\alpha^i\}_{i\in\omega}\right)M\right)_A \\
= \quad & \{\text{ composition of functors and natural transformations }\} \\
& \left(\sqcup\{\alpha^i\}_{i\in\omega}\right)_{M.A} \\
= \quad & \{\text{ least upper bounds are computed pointwise }\} \\
& \sqcup\left\{\alpha^i_{M.A}\right\}_{i\in\omega} \\
= \quad & \{\text{ composition of functors and natural transformations }\} \\
& \sqcup\left\{\left(\alpha^i M\right)_A\right\}_{i\in\omega} \\
= \quad & \{\text{ least upper bounds are computed pointwise }\} \\
& \left(\sqcup\{\alpha^i M\}_{i\in\omega}\right)_A
\end{aligned}$$

for any A in $\mathbb{B}$.

To summarize, we have proved:

Lemma 3.22: A higher-order functor K is locally ω-continuous if it satisfies any of the following:

(i) K is an identity functor;

(ii) K is a constant functor;

(iii) K is a composite functor ML where both M and L are locally ω-continuous;

(iv) K is a lifted functor $\dot{F}$ for some locally ω-continuous functor F (including bifunctors);

(v) K is of the form $L \ddot{\otimes} M$ where L, M and $\dot{\otimes}$ are all locally ω-continuous functors;

(vi) K is a precomposition functor $(\circ M)$ for some functor M (M need not be locally ω-continuous).

So clearly our base functor for nests:

$$N \overset{\mathit{def}}{=} \underline{\underline{\mathbb{1}}} \mathbin{\ddot{+}} \underline{\mathit{Id}_{\mathbb{C}}} \mathbin{\ddot{\times}} (\circ P)$$

is locally ω-continuous provided that $\dot{+}$ and $\dot{\times}$ are.

3.5. Chapter summary

Folding over a value of a non-uniform type may involve replacing infinitely many different instances of the type's polymorphic constructors. One method of doing this is by designing a fold combinator that takes *polymorphic* arguments which can then be used at appropriate instances to replace each constructor instance. Such a combinator corresponds to a catamorphism operator at the level of *functor categories*.

Regular parameterized types give rise to initial algebras in functor categories. Importantly, *linear non-uniform* data-types can be expressed as initial algebras in functor categories, and the resulting catamorphism operators correspond to fold combinators that take polymorphic arguments.

Catamorphisms in functor categories suffer expressive limitations as a result of being natural transformations — only operations that are parametrically polymorphic can be expressed in this way, thus excluding many useful and desirable operations such as summing a structure of numbers or, more fundamentally, *mapping* a function across a structure.

To prove the existence of initial algebras in functor categories, one can use the same techniques as described in chapter 2. For the most part, the results lift straightforwardly to functor categories, however some extra care is required with regard to strictness in the order-enriched case.

To deal with non-uniform data-types in this way it is necessary to consider not only polynomial or regular functors, but also those using precomposition functors $(\circ M)$ which introduce the modifying functors. Although we give only a weak result about the ω-cocontinuity of $(\circ M)$ when M is a right adjoint, in the order-enriched setting it is easy to prove that $(\circ M)$ is *locally ω-continuous* for an *arbitrary* modifying functor M.

The main contribution of this chapter is the lifting of the standard results about the existence of initial algebras to functor categories. However, during the final stages of this thesis, I became aware that several researchers had been using similar techniques for solving domain equations in order-enriched functor categories in order to give denotational semantics to λ-calculi extended with subtyping [FiechS94, FiechH94, Tsuiki95].

Chapter 4. Folding with algebra families

In this chapter we look at an alternative approach to folding over non-uniform data-types. Why do we need another approach? We have seen that folds in functor categories, although a natural extension of the standard folds in categories, do not allow us to write many of the functions that we would wish because of the excessive polymorphism. We would prefer to find a more effective or expressive recursion pattern, and develop a theory that more closely matches the programs that we write.

4.1. Overview of the process

To begin, we look for inspiration in the operational behaviour of folds. Recall that folds over uniform data-types recursively replace constructor functions in a data-value with the corresponding arguments to the fold. Functor category folds operate in this way as well, although they replace *instances of polymorphic constructors* with *instances from the polymorphic arguments to the fold.*

In this chapter we develop an approach to folding over non-uniform data-types based on the following steps:

(i) analyze the type definition to identify *exactly* which instances of polymorphic constructor functions could possibly occur in a data-value;

(ii) build a structure that contains replacements for each possible instance;

(iii) recursively traverse the data-value replacing the constructor instances.

This can be seen as a logical extension of the traditional process of folding over uniform types. Take lists, for example,

```
data List a = NilL
            | ConsL a (List a)
```

Then for a data-value of type (List Int), step (i) above is trivial — the only instances of the constructor functions that can occur in such a value are the Int instances:

```
NilL  :: List Int
ConsL :: Int → List Int → List Int
```

For step (ii) then, we need only supply two replacements — one for NilL_{Int} and one for $\text{ConsL}_{\text{Int}}$ — and these can easily be given as arguments to a fold combinator that performs step (iii).

For data-types defined using non-uniform recursion, complications arise because the number of instances identified in step (i) will, in general, be infinite, and this necessitates providing an *infinite* structure of replacement functions. Of course we can never actually compute an entire infinite data-structure, so we exploit laziness to evaluate only the parts of the structure we require. These infinite structures we will call *algebra families*, and the combinators we design to replace constructor instances with components from an algebra family we will call *algebra family folds*.

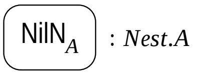
In the following sections we will look at each step in detail and describe how to code the resulting data-structures and fold combinators in Haskell. The second half of the chapter is concerned with constructing an appropriate categorical semantics for the algebra families and folds we describe in the first half. We give a semantics in terms of initial algebras, so the theory will be familiar from the standard theory of folds over uniform types and also the higher-order folds we presented in chapter 3. We can then reformulate the calculational laws that stem from initial algebras for the case of our algebra families.

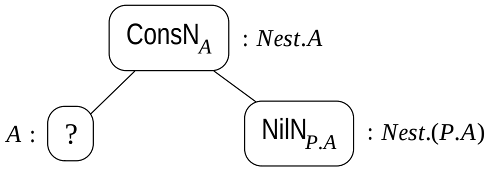
4.2. Identifying the possible constructor instances

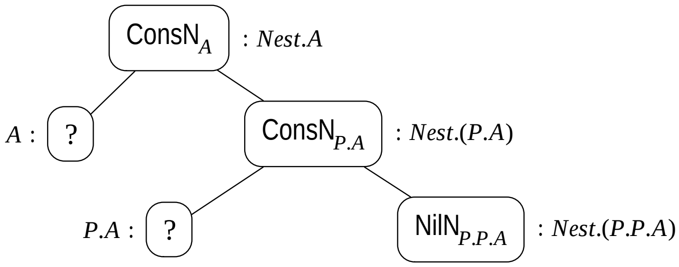
The first step is to analyze our given non-uniform data-type definition to see which instances of the constructor functions might occur in a value of our data-type. Let us look at the nest data-type as an example:

```
data Nest a = NilN
            | ConsN a (Nest (P a))
```

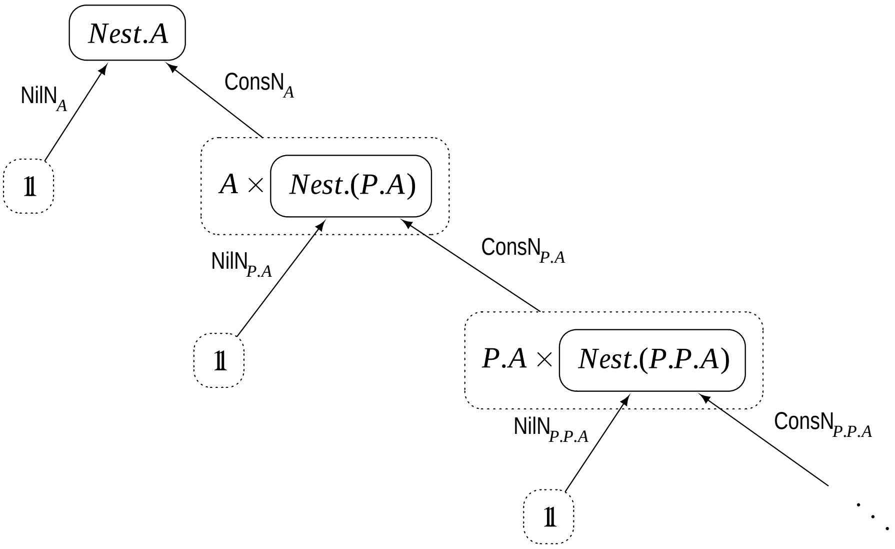
For a fixed type A, consider the ways that a value of type $Nest.A$ may be constructed. We can enumerate the possible structures by the number of constructors they contain. The first three are shown below:







We can represent these possibilities more compactly with a different kind of branching diagram:



Each path in the diagram corresponds to one of the structures that we started to draw above. From this diagram we can read off which constructor instances may occur in a value of *Nest.A* — NilN and ConsN may occur at the instances *A*, *P.A*, *P.P.A*, and so on.

The type *Nest* has only one recursive call, and it is modified by one functor, *P*. What happens if we introduce a new recursive call, with a different modifying functor, say *Q*?

```
data T a = N
         | C a (T (P a))
         | D a (T (Q a))
```

We have placed the new recursive call in a new constructor, but there is no reason why we couldn't have extended the original ConsN constructor with a second recursive call. If we work out the possible paths of constructor instances for T we get the ternary tree structure in figure 4.1. We let the tree grow horizontally this time for reasons of space. We see that each constructor may appear at the instances *A*, *P.A*, *Q.A*, *P.P.A*, *P.Q.A*, *Q.P.A*, *Q.Q.A*, and so on. In fact, any finite string made up of the func-

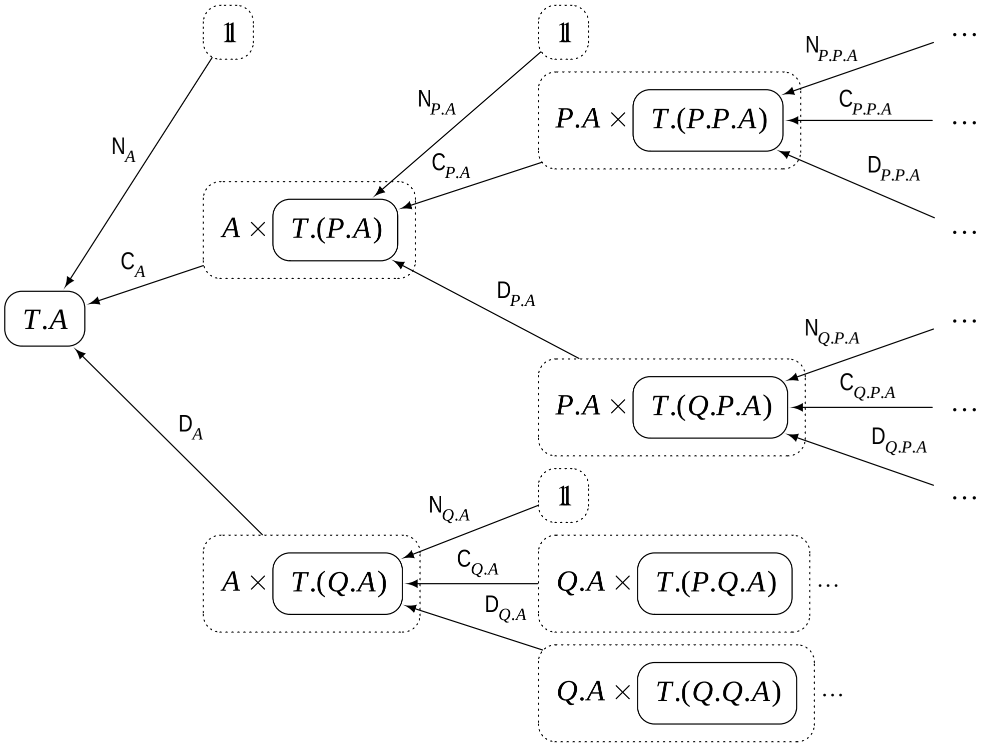


Figure 4.1. Constructor instances for a data-type with two modifying functors

tors P and Q applied to A will appear as an instance in the above diagram if we follow a long enough path. Moreover, these are the *only* instances that may occur, and are therefore the only constructor instances that we need contemplate replacing.

In order to give a general description of how to identify the possible constructor instances that may occur in a value of a given non-uniform data-type we will make three assumptions about the type definition:

(i) every recursive call is non-uniform;

(ii) for each recursive call, the name of the modifying functor is distinct from the names of the other modifying functors;

(iii) none of the modifying functors are the same as the type being defined, that is, we do not consider *non-linear* non-uniform recursion.

The first two assumptions are for convenience when we come to describe the semantics for our algebra families, because, as we will see, it is natural to index the components of our algebra families by finite strings of the *names* of our modifying functors, and for this to work we need that each has a distinct name. The assumptions prevent type definitions such as

```
data T a = N a
         | C (T a)
         | D (T (P a)) (T (P a))
```

which violates both (i) and (ii) — the first recursive call is uniform and violates (i), the second and third recursive calls both have P as their modifying functor, and thus violate (ii). We can get around these restrictions by:

(i) considering uniform recursive calls such as (T a) to be non-uniformly recursive with Id as the modifying functor, so (T a) becomes (T (Id a));

(ii) if multiple recursive calls use the same modifying functor then just define a new copy of the modifying functor but give it a different name.

Applying both of these techniques to our data-type above we arrive at the acceptable type definition:

```
data Id a = Id a
data P_1 a = P a
data P_2 a = P a
data T' a = N' a
          | C' (T' (Id a))
          | D' (T' (P_1 a)) (T' (P_2 a))
```

It is possible to avoid making these assumptions if we adopt a different indexing scheme in our semantics, for example, if we assign each recursive call a unique number and then index our algebra families by finite strings of those numbers, but this would lead to a more complicated exposition of the semantics.

The third assumption — that the types do not involve non-linear non-uniform recursion — is required because the semantics we give later is designed only for linear non-uniform recursion.

Given a type definition of the form

$$\begin{aligned}\underline{\mathsf{data}}\ \mathsf{T\ a}\ =\ & \mathsf{C}_1\ \mathsf{E}_1(\mathsf{a},\ \mathsf{T}\ (\mathsf{M}_1\ \mathsf{a}),\ldots,\ \mathsf{T}\ (\mathsf{M}_m\ \mathsf{a}))\\ & \ldots\\ & \mathsf{C}_n\ \mathsf{E}_1(\mathsf{a},\ \mathsf{T}\ (\mathsf{M}_1\ \mathsf{a}),\ldots,\ \mathsf{T}\ (\mathsf{M}_m\ \mathsf{a}))\end{aligned}$$

and satisfying the assumptions described above, then the constructor instances that can occur in a value of type `T A` for some type A is given by the set:

$$\left\{ \left(\mathsf{C}_i\right)_{\Omega.A} \mid 1 \le i \le n,\ \Omega \in \left\{ M_1, \ldots, M_m \right\}^* \right\}$$

where $\left\{ M_1, \ldots, M_m \right\}^*$ is the free monoid on $\left\{ M_1, \ldots, M_m \right\}$ or the set of finite strings of modifying functor names, treating the empty string as *Id*.

4.3. Algebra families

Now that we have identified the instances of the constructors that we may need to replace, we can set about providing replacements for them. We will somehow need to pass these replacements as arguments to a fold combinator, so the first problem is to define a new data-type to store the replacements for the constructors. In a non-uniform data-type there will be potentially infinitely many constructor instances to replace, and, accordingly, we will need an infinite data-structure to hold the replacements if we are to provide a replacement for each instance. Fortunately lazy languages such as Haskell are quite at home with infinite data-structures because they only evaluate parts of such structures on demand.

Let us look more closely at the types of the constructor instances, and the possible types of functions to replace them. We start by looking at the simple case of the Nest data-type which has only a single modifying functor P. We know that the constructors to be replaced in a value of type $Nest.A$ may be at the instances A, $P.A$, $P.P.A$, and so on. Furthermore, we know that the constructor instances will be encountered in that order, and we will use this fact to simplify storage of the replacement functions.

Consider how a higher-order catamorphism would replace the constructor instances in a nest. The arguments to the catamorphism are two natural transformations

$$\lambda : \underline{\mathbb{1}} \xrightarrow{\bullet} R \qquad\qquad \rho : Id_{\mathbb{C}} \dot{\times} RP \xrightarrow{\bullet} R$$

So the catamorphism $(\!|\, \lambda \dot{\triangledown} \rho \,|\!)$ would replace

$$\mathsf{NilN}_B : \mathbb{1} \to Nest.B$$

with

$$\lambda_B : \mathbb{1} \to R.B$$

and

$$\mathsf{ConsN}_B : B \times Nest.P.B \to Nest.B$$

with

$$\rho_B : B \times R.P.B \to R.B$$

for any type B in the base category. In general, if we write P^n to mean n applications of the endofunctor P, then $(\!|\, \lambda \dot{\triangledown} \rho \,|\!) : Nest \xrightarrow{\bullet} R$ replaces constructors at the instance $P^n.A$ with the $P^n.A$ instance of the appropriate argument to the catamorphism. In this way, $(\!|\, \lambda \dot{\triangledown} \rho \,|\!)$ recursively transforms $Nest$ structures into R structures. We can define a data-type to store only the instances of the catamorphism argument that might be necessary:

```
data NestAlgFam r a =
    MkNestAlgFam {
      nilnreplace :: r a,
      consnreplace :: a → r (P a) → r a,
      next :: NestAlgFam r (P a)
    }
```

For those not familiar with Haskell's record syntax, this is an alternative syntax for defining product types that is convenient for working with records. Instead of defining *constructor* functions, nilnreplace, consnreplace and next are *selector* functions that select the appropriate field from a record. They have types:

```
 nilnreplace :: NestAlgFam r a → r a
consnreplace :: NestAlgFam r a → (a → r (P a)) → r a
        next :: NestAlgFam r a → NestAlgFam r (P a)
```

The mechanism is just a convenient way of assigning meaningful names to the projection functions from a product. Records are defined by setting their fields in the following way

```
algFam :: NestAlgFam r a
algFam = MkNestAlgFam {
             nilnreplace = … ,
             consnreplace = … ,
             next = …
         }
```

A NestAlgFam structure contains three fields — a replacement for the NilN constructor, a replacement for the ConsN constructor, and the next field which is another NestAlgFam structure that contains the replacements for the next instances of the constructors. A NestAlgFam structure can contain exactly the instances of the higher-order catamorphism arguments that could be needed to replace the constructors in a nest, and they are stored in the order that they would occur.

Looking at the types of the replacement functions we see that we have abstracted away from the Nest type constructor to the type constructor variable r. There are further abstractions of the types that we can make that will allow us greater flexibility in the folds we can define. We can abstract the parameter type a inside nest structures to a new type variable b. Thus we get the following more general variant of algebra families for nests:

```
data NestAlgFam r a b =
        MkNestAlgFam {
          nilnreplace :: r b ,
          consnreplace :: a → r (P b) → r b ,
          next :: NestAlgFam r s (P a) (P b)
        }
```

Note that we can only abstract occurrences of the parameter type a that occur *inside* the parameter to a nest type, so that although ConsN has type

```
ConsN :: a → Nest (P a) → Nest a
```

its replacement may not have type

```
consnreplace :: b → r (P b) → r b
```

but instead may have the type

```
consnreplace :: a → r (P b) → r b
```

The reason for this is that our fold combinator will recursively process the (Nest (P a)) structure, transforming it into a (r (P b)) structure, but there will be no such processing of the a value, so its type will remain unchanged.

We can also abstract away from the modifying functor P to another type constructor variable which we call s for the “step” functor. This last abstraction leads to the most general algebra family type that we will consider:

```
data NestAlgFam r s a b =
       MkNestAlgFam {
         nilnreplace :: r b ,
         consnreplace :: a → r (s b) → r b ,
         next :: NestAlgFam r s (P a) (s b)
       }
```

Note that the type definitions of our various algebra families are themselves non-uniformly recursive — this is to be expected because they must store infinitely many replacement functions, all with differing types. Consequently, the construction of algebra families is by no means trivial, and we consider the problem in more detail in section 4.5.

4.3.1. Data-type definitions with multiple recursive calls

So far we have considered only the case of replacing the constructors in nests. Let us look at what happens when a data-type definition contains more than one recursive call. Recall the type

```
data T a = N
         | C a (T (P a))
         | D a (T (Q a))
```

from section 4.2. What would the algebra family look like? The constructors have types:

```
N :: T a
C :: a → (T (P a)) → T a
D :: a → (T (Q a)) → T a
```

and we know that an algebra family must contain replacements for these, say

```
data TAlgFam a = TAlgFam {
          nreplace :: T a,
          creplace :: a → (T (P a)) → T a,
          dreplace :: a → (T (Q a)) → T a,
          …
     }
```

What about the next function, or in this case, functions. Looking back at the diagram 4.1 on page 111, we see that the two recursive calls in the type definition of T lead down two different branches of the tree structure. To capture this multiple-branching structure in our algebra family type, we must introduce *two* next functions — one for moving down the first branch, and one for moving down the second:

```
nextP :: TAlgFam (P a)        nextQ :: TAlgFam (Q a)
```

Of course, the type of algebra families we have described here is not flexible enough to allow us to do anything useful — as we did with nests, we must abstract the types of the constructors. As with nests, we can abstract the type constructor T to a "result" functor r, and introduce a new type variable b to take the place of the type a inside recursively constructed values. As for the modifying functors P and Q, we can introduce *two* "step" functors, call them s and u, to give the general type of algebra families for T:

```
data TAlgFam r s u a b =
        MkTAlgFam {
            nreplace :: r b,
            creplace :: a → (r (s b)) → r b,
            dreplace :: a → (r (u b)) → r b,
            nextP :: TAlgFam r s u (P a) (s b),
            nextQ :: TAlgFam r s u (Q a) (u b)
        }
```

Notice the difference in the types of the two families produced by nextP and nextQ. The nextP and nextQ selectors are used by the fold function to choose the appropriate branch to follow.

4.3.2. Specialized families of algebras

We have described how to capture families of replacement functions with very general types by introducing “result” and “step” functor variables. For many applications, however, this high level of generality is unnecessary, and some or all of the result or step functors can be set to be the identity functor. In these cases, we can provide simpler “specialized” versions of the structures that hold replacement functions. Consider again the type of nest algebra families:

```
data NestAlgFam r s a b =
        MkNestAlgFam {
          nilnreplace :: r b ,
          consnreplace :: a → r (s b) → r b ,
          next :: NestAlgFam r s (P a) (s b)
        }
```

If we wanted to capture families in which both the result and step functors were the identity, for example, then we could use the simpler type:

```
data NestAlgFam’ a b =
        MkNestAlgFam’ {
          nilnreplace’ :: b ,
          consnreplace’ :: a → b → b ,
          next’ :: NestAlgFam’ (P a) b
        }
```

All we have done is to omit any mention of the type constructors r and s. Of course, the fold function must now have a specialized type:

```
foldNestAlgFam' :: NestAlgFam' a b → Nest a → b
```

but the definition of the fold function does not change — it still just replaces constructor functions.

One of the reasons for simplifying the types in this way is that it makes the process of *constructing* algebra families easier — it becomes easier to define helpful combinators that produce algebra families, as will be seen in the next section.

For many common applications that involve collapsing the non-uniform structure, for example, summing a nest, these specialized algebra families will suffice, and are more convenient for programming. Applications that require some or all of the non-uniform structure to be preserved require the more general algebra families. The prime example of this is the map operation which we will look at in detail in chapter 6. A map must preserve *all* of the structure of a data-value and only change the values contained within the structure.

4.4. Algebra family folds

We now show how to define a combinator to recursively process a nest structure, replacing constructor instances with the corresponding functions contained in an algebra family. In other words, we will define an *algebra family fold*.

Suppose we have an algebra family of type (NestAlgFam r s a b), and we wish to use it to fold over some value of type (Nest a). We need a combinator

```
foldNestAlgFam :: NestAlgFam r s a b → Nest a → r b
```

to perform the replacement and produce a result value of type (r b). All the combinator does is replace constructors, and can be simply defined as:

```
    foldNestAlgFam fam NilN = (nilnreplace fam)
    foldNestAlgFam fam (ConsN x xs) =
             (consnreplace fam) x (foldNestAlgFam (next fam) xs)
```

The selector functions nilnreplace and consnreplace select the appropriate replacement function from the family fam. Note that in the recursive call to foldNestAlgFam, the algebra family used is (next fam). This has the effect of discarding the current replacements, because they will no longer be needed, and moving one level down in the tree structure that holds the replacement functions. We can see how this fold operates on an example nest:

```
   foldNestAlgFam fam (ConsN (7,8) (ConsN 12 NilN))
=
   (consnreplace fam) (7,8) (foldNestAlgFam (next fam) (ConsN 12 NilN))
=
   (consnreplace fam) (7,8)
       ((consnreplace (next fam)) 12
           (foldNestAlgFam (next (next fam)) NilN))
=
   (consnreplace fam) (7,8)
       ((consnreplace (next fam)) 12
           (nilnreplace (next (next fam))))
```

If we compare this with the original nest then we see that the first ConsN constructor has been replaced by (consnreplace fam), the second by (consnreplace (next fam)), and the final constructor NilN by (nilnreplace (next (next fam))). In other words, the constructors have been replaced by the corresponding functions from the correct level of the tree structure.

In the case of data-types with multiple recursive calls, the only complication is to choose the correct branch of the algebra family to follow by using the next selectors. For example, for our data-type

```
    data T a = N
```

```
                | C a (T (P a))
                | D a (T (Q a))
```

from section 4.3.1. The fold combinator is

```
foldTAlgFam :: TAlgFam r s u a b → Nest a → r b
foldTAlgFam fam N = (nreplace fam)
foldTAlgFam fam (C x xs) =
     (creplace fam) x (foldTAlgFam (nextP fam) xs)
foldTAlgFam fam (D x xs) =
     (dreplace fam) x (foldTAlgFam (nextQ fam) xs)
```

When recursively processing the first recursive call to T, which is modified by the functor P, the fold combinator uses the nextP selector to move down the "P" branch in the algebra family. For the second call, modified by Q, it uses the nextQ selector.

4.5. Constructing algebra families for nests

We now know how to define the types to hold our algebra families, but we have not considered the task of actually building a value of such a type. For simplicity we will use the specialized type of algebra families in which all the type constructor variables have been fixed to be the identity:

```
data NestAlgFam' a b =
        MkNestAlgFam' {
           nilnreplace' :: b ,
           consnreplace' :: a → b → b ,
           nextP' :: NestAlgFam' (P a) b
        }
```

This is an infinite data-structure, and there is no way that we can hope to fill it by listing all the values explicitly. What we need are some "seed" values to start us off, and a way to "grow" the rest of the tree structure from those seeds. Suppose we are given as seeds values for the first two replacement functions:

```
nilnreplace' :: b
consnreplace' :: a → b → b
```

then we wish to construct the two replacement functions at the next level in the tree structure. We reach the next level by using the nextP' selector, which gives a (NestAlgFam' (P a) b) structure. Therefore, the replacements at the next level must have type:

$$\mathsf{nilnreplace'}_{\mathsf{P}} :: \mathsf{b}$$

$$\mathsf{consnreplace'}_{\mathsf{P}} :: (\mathsf{P}\ \mathsf{a}) \rightarrow \mathsf{b} \rightarrow \mathsf{b}$$

We subscript them with P to distinguish them from the replacements at the top level. We somehow need to grow these values from the values given as seeds for the top-level replacements. The type for $\mathsf{nilnreplace'}_{\mathsf{P}}$ has not changed, and we can therefore use the same value as nilnreplace'. However, the type of $\mathsf{consnreplace'}_{\mathsf{P}}$ has introduced a new P constructor. If we write the types categorically, then we somehow have to grow

$$\mathsf{consnreplace}'_P : P.A \times B \rightarrow B$$

from the top-level seed

$$\mathsf{consnreplace}' : A \times B \rightarrow B$$

Suppose we have a function collapseP with type $P.A \rightarrow A$, then we could construct $\mathsf{consnreplace}'_P$ as:

$$\mathsf{consnreplace}'_P \overset{def}{=} P.A \times B \xrightarrow{\mathsf{collapseP} \times Id} A \times B \xrightarrow{\mathsf{consnreplace}'} B$$

This process works also for moving to deeper levels of the tree. For instance, the next level requires

$$\mathsf{consnreplace}'_{PP} : P.P.A \times B \to B$$

and this can be constructed from $\mathsf{consnreplace}'_{P}$ as

$$\mathsf{consnreplace}'_{PP} \stackrel{def}{=} P.P.A \times B \xrightarrow{P.\mathsf{collapseP} \times Id} P.A \times B \xrightarrow{\mathsf{consnreplace}'_{P}} B$$

where this time we must *map* collapseP across the P structure. We can continue in this way and generate the entire infinite tree structure.

We can capture the whole process in a combinator, that takes as arguments the seed values of the top-level replacement functions, and also the collapseP function:

```
generateNestAlgFam' :: b →
                             (a → b → b) →
                                  (P a → a) →
                                       NestAlgFam' a b
generateNestAlgFam' n c collapseP =
  MkNestAlgFam' {
    nilnreplace' = n ,
    consnreplace' = c ,
    nextP' = generateNestAlgFam'
                 n
                 (\ x → c (collapseP x))
                 (mapP collapseP)
  }
```

The combinator fills in the given seed values n and c, and generates the seeds for the next level using collapseP. It also applies mapP to collapseP to give it the correct type for the next level.

The generateNestAlgFam' combinator is suitable for defining the algebra families for a large variety of possible folds over nests. As an example, we return to the problem that defeated higher-order catamorphisms in section 3.3.2 — summing a nest of integers. We would like a function

```
sumNest :: Nest Int → Int
```

to be expressed as an algebra family fold. Using the simplified algebra families, and the generateNestAlgFam' combinator described above, we see that we must supply as ingredients two seeds of type Int and (Int → Int → Int), and a function for collapsing pairs of integers with type (P Int → Int). The first seed will be the result of summing an empty nest, and should therefore be zero. The second seed combines the current integer value with the result of recursively processing the rest of the nest, and should be the binary addition operator. To collapse pairs of integers we just sum them. We arrive at the definition:

```
sumNest = foldNestAlgFam' algfam
  where sumPair (x,y) = x + y
        algfam = generateNestAlgFam' 0 (+) sumPair
```

and this correctly sums nests.

It is possible to define a combinator like generateNestAlgFam' for the more general nest algebra families in which the result and step functors are non-identity, but in our experience the extra type constructors are a hindrance and can often be avoided.

In contrast to the fold combinators of Bird and Paterson [BirdP99], we have taken the conscious decision to separate the two processes of constructing the replacement functions and then actually using them to replace constructor instances because we believe it simplifies the exposition. However, this does lead to more long-winded programs. It is possible to hide this separation from the programmer and combine the two phases into a single combinator:

```
foldNest :: b →
```

```
                  (a → b → b) →
                      (P a → a) →
                          Nest a → b
    foldNest e f g = foldNestAlgFam' fam
       where fam = generateNestAlgFam' e f g
```

This combinator could be further transformed to eliminate the intermediate algebra family type altogether. The result is a combinator with a simple type, not dissimilar to that of the familiar fold over lists, that would be more appropriate for the programmer who is unconcerned with the underlying principles.

In general, for non-uniform data-types, the types of the algebra families are rather broad and a correct candidate algebra family for a desired operation is often not obvious at first sight. There are at least two approaches to the problem:

(i) the programmer can work out by hand what the first few members of a family should be and then try to discover a pattern between the members that can be used to produce the rest of the family;

(ii) the data-type designer can try to predict the needs of the programmer and restrict the size of the search space in helpful ways by

  - simplifying the algebra family types;
  - providing general-purpose combinators that capture common patterns used to produce algebra families.

The first approach is deficient in several ways:

- it relies on guesswork or insight on the part of the programmer;
- it somewhat defeats the point of providing structured recursion combinators because the programmer must anyway invent a polymorphic recursive

function over a non-uniform type (the algebra family type) in order to produce the desired algebra family with which to fold.

Clearly the programmer should be given some assistance in constructing algebra families, and this is an obvious area for future work (see section 7.3.1).

We have already mentioned the merits of simplifying the algebra family types — many operations do not need the full generality of the types, and constructing algebra families with simplified types then becomes easier. This does not mean that we can get by with simplified algebra family types alone — some operations, specifically those that need to retain or transform the non-uniform structure of a value, will require the more general types.

4.5.1. Algebra family folds generalize functor category catamorphisms

In this section we show that any functor category catamorphism on nests can be performed as an algebra family fold. The arguments to a functor category catamorphism operator are natural transformations or polymorphic

$$\lambda : \underline{\mathbb{1}} \underset{\cdot}{\longrightarrow} R$$

and

$$\rho : Id_{\mathbb{C}} \mathbin{\dot{\times}} RP \underset{\cdot}{\longrightarrow} R$$

for some result functor R. In Haskell we treat these as functions as polymorphic types:

```
lam :: ∀ a . r a
```

and

```
rho :: ∀ a . a → r (P a) → r a
```

The trick is to use these polymorphic arguments to build a suitable algebra family to give as the argument to our algebra family fold combinator. The function lam and rho are fully polymorphic in that they are families containing an instance for *every* type. We are only interested in a subset of those types — to fold over a nest of some type b, we only require the instances b, (P b), (P (P b)), and so on. It is straightforward to define a combinator that will extract these instances from polymorphic functions and place them in an algebra family:

```
nestPolyArgsToAlgFam :: (∀ a . r a) →
                                (∀ a . a → r (P a) → r a) →
                                  NestAlgFam r P b b
nestPolyArgsToAlgFam lam rho =
  MkNestAlgFam {
    nilnreplace = lam ,
    consnreplace = rho ,
    nextP = nestPolyArgsToAlgFam lam rho
  }
```

The line

```
nilnreplace = lam
```

chooses the correct instance from the polymorphic function lam to fill the nilnreplace field at each level of the algebra family. Similarly for consreplace. The functor category catamorphism $(\!|\, \lambda \mathbin{\dot{\triangledown}} \rho \,|\!)$ can then be simulated with an algebra family fold:

```
foldNestAlgFam (nestPolyArgsToAlgFam lam rho)
```

4.6. Deriving algebra families and folds for other non-uniform data-types

The general form of a non-uniform data-type definition in Haskell is:

```
data T a = C¹ E₁
         | ...
         | Cⁿ Eₙ
```

Where E_1 to E_n are expressions describing the source type of each constructor. The source type of a constructor can include the parameter a, and recursive calls to T with the parameter a modified by a modifying functor M_1 to M_m, where m is the number of modifying functors. To make this explicit, we write

$$E_i(a,\ T(M_1\ a),\ \ldots,\ T\ (M_m\ a))$$

Remember that we are assuming every recursive call is non-uniform, so this prevents (T a) from appearing in the E_i's. The target type of each constructor is (T a), so we can write the types of the constructors in full as:

$$C^i :: E_i(a,\ T\ (M_1\ a),\ \ldots,\ T\ (M_m\ a)) \rightarrow T\ a$$

If we take the Nest data-type as an example then we have two constructors, so n is 2, and one modifying functor, P, so m equals 1. The first constructor, C^1, is NilN, and the second constructor is ConsN. Written categorically, the E_i's determine the source types of the constructors, so, for nests,

$$E_1(A, Nest.P.A) = \mathbb{1}$$

because NilN_A is a constant, and

$$E_2(A, Nest.P.A) = A \times Nest.P.A$$

because ConsN_A takes these two arguments.

To construct the algebra family type for a data-type, we must provide replacements for each constructor:

$$c^1replace :: \ldots$$

$$\ldots$$

$$c^{n}\mathsf{replace} :: \ldots$$

but we can abstract the types of these replacements in three ways:

(i) abstract the type constructor T to a type constructor variable r — the "result" functor;

(ii) abstract each modifying functor $\mathsf{M_i}$ to a type constructor variable $\mathsf{s_i}$ — the "step" functors;

(iii) introduce a new type variable b to take the place of the parameter type a inside recursive calls.

If we apply these three abstractions then each constructor replacement can have type:

$$c^{i}\mathsf{replace} :: \mathsf{E_i(a,\ r\ (s_1\ b), \ldots,\ r\ (s_m\ b))} \rightarrow \mathsf{r\ b}$$

We also have to provide the "next" functions for moving down the appropriate branches of the tree structure. For each modifying functor $\mathsf{M_i}$, we will need a new algebra family with the parameter a modified by $\mathsf{M_i}$, and b modified by the corresponding step functor, $\mathsf{s_i}$:

$$\mathsf{nextM_i} :: \mathsf{TAlgFam\ r\ s_1\ \ldots\ s_m\ (M_i\ a)\ (s_i\ b)}$$

The record can then be written out in full as:

$$
\begin{array}{l}
\underline{\mathsf{data}}\ \mathsf{TAlgFam\ r\ s_1\ \ldots\ s_m\ a\ b} = \\
\qquad\quad \mathsf{MkTAlgFam}\ \{ \\
\qquad\qquad c^{1}\mathsf{replace} :: \mathsf{E_1(a,\ r\ (s_1\ b), \ldots,\ r\ (s_m\ b))} \rightarrow \mathsf{r\ b}\ , \\
\qquad\qquad\quad \ldots \\
\qquad\qquad c^{1}\mathsf{replace} :: \mathsf{E_1(a,\ r\ (s_1\ b), \ldots,\ r\ (s_m\ b))} \rightarrow \mathsf{r\ b}\ , \\
\\
\qquad\qquad \mathsf{nextM_1} :: \mathsf{TAlgFam\ r\ s_1\ \ldots\ s_m\ (M_1\ a)\ (s_1\ b)}, \\
\qquad\qquad\quad \ldots
\end{array}
$$

```
      nextM_m :: TAlgFam r s_1 ... s_m (M_m a) (s_m b)
    }
```

This gives the most general type of algebra families we have described.

Simpler, specialized versions can be produced by fixing some or all of the result and step functors to be the identity. For example, if we fix all the step functors to be the identity, but leave the result functor then we arrive at the specialized type:

```
data TAlgFam r a b =
    MkTAlgFam {
      c^1replace :: E_1(a, r b,..., r b) → r b ,
        ...
      c^1replace :: E_1(a, r b,..., r b) → r b ,

      nextM_1 :: TAlgFam r (M_1 a) b ,
        ...
      nextM_m :: TAlgFam r (M_m a) b
    }
```

The fold combinator to perform the constructor replacements is easy to define. It has type

```
foldTAlgFam :: TAlgFam r s_1 ... s_m a b → T a → r b
```

and replaces each constructor with the corresponding replacement from the algebra family,

```
foldTAlgFam fam (C^i ...) = (c^ireplace fam) ...
```

The only complication is choosing the correct subfamily to pass when recursively processing the arguments to the constructor. If the j-th argument to the constructor C^i is of type $(T\ (M_k\ a))$, then it must be recursively processed with the algebra family that results from applying nextM_k to the current family, so

```
foldTAlgFam fam (C^i ... x_j ...) =
    (c^i replace fam) ... (foldTAlgFam (nextM_k fam) x_j) ...
```

For specialized algebra families, the fold combinator is defined in exactly the same way.

4.7. Examples

Our major example of algebra family folds will appear in chapter 6 where they are used to define and prove properties about maps over non-uniform data-types.

4.7.1. Folding sequences as if they were lists

Recall the definition of Okasaki's [Okasaki98] *binary random-access lists*, or *sequences*:

```
type P a = (a,a)
data Seq a = NilS
           | ZeroS (Seq (P a))
           | OneS a (Seq (P a))
```

This data-type is used to implement lists when efficient look-up and update operations are required. However, in moving from lists to sequences we lose the ability to use the traditional fold combinator for lists:

```
foldList :: b →
            (a → b → b) →
            (List a → b)
```

which is used so often to define functions on lists. The sequence type is non-uniform, and we cannot define a corresponding traditional fold for sequences. Furthermore, we cannot use higher-order catamorphisms because we encounter the same problems with polymorphism as we did when trying to sum a nest of integers — we can easily sum a *list* of integers with a traditional fold, but there is no way to sum a *sequence* of integers with a higher-order catamorphism.

One solution would be to define a function that converts a sequence into a list:

```
seqToList :: Seq a → List a
```

by "collapsing" the extra structure in the sequences, and then composing the appropriate list fold:

$$\begin{tikzcd}\mathsf{Seq\ a} \arrow[r, "\mathsf{seqToList}"] & \mathsf{List\ a} \arrow[r, "\mathsf{(foldList\ e\ f)}"] & \mathsf{b}\end{tikzcd}$$

The disadvantage of this approach is that we construct an intermediate data-structure of lists.

We develop another approach, using algebra family folds, that does not produce an intermediate data-structure. We will use the simplified algebra families, where the "result" and "step" functors are taken to be the identity. The type of sequences is just an extension of the type of nests, and, as we might expect, the algebra families and fold function are just an extension of those for nests (modulo renaming of constructors). First we give the simplified algebra family type:

```
data SeqAlgFam' a b =
      MkSeqAlgFam' {
         nilsreplace' :: b ,
         zerosreplace' :: b → b ,
         onesreplace' :: a → b → b ,
         nextP' :: SeqAlgFam' (P a) b
      }
```

Although the type of sequences contains *two* recursive calls, they are both with the modifying functor P. For the particular application we are considering, the algebra families we build will be identical no matter which of the two branches we follow, so we may as well identify the two branches into a single branch. Thus we only need a single "next" selector in the above type of algebra families. This is another example of how the algebra family types can be specialized for convenience. The fold function is defined to take this into account:

```
foldSeqAlgFam' :: SeqAlgFam' a b → Seq a → b
foldSeqAlgFam' fam NilS = (nilsreplace' fam)
foldSeqAlgFam' fam (ZeroS ps) =
        (zerosreplace' fam) (foldSeqAlgFam' (nextP' fam) ps)
foldSeqAlgFam' fam (OneS x ps) =
        (onesreplace' fam) x (foldSeqAlgFam' (nextP' fam) ps)
```

(Compare these with the algebra family and fold function for nests.) We can use a standard function for generating algebra families for sequences:

```
generateSeqAlgFam' ::
   b → (b → b) →
      (b → b) → ((b → b) → (b → b)) →
         (a → b → b) → (∀ c . (c → b → b) → (P c → b → b)) →
            SeqAlgFam' a b
generateSeqAlgFam' n nn z nz o no =
  MkSeqAlgFam' {
    nilsreplace' = n ,
    zerosreplace' = z ,
    onesreplace' = o ,
    nextP' = generateSeqAlgFam' (nn n) nn (nz z) nz (no o) no
  }
```

in which we give the seeds for the family, and polymorphic functions with which to grow them. Note that the types of the nn and nz arguments (the “growing” functions) should really be universally quantified like the type of the argument no. However, because the polymorphic argument does not actually appear in the type, the functions are essentially monomorphic and the Hugs type-checker complains if we try to give it a quantifier.

Now, suppose we wish to sum a sequence of integers. If we were working with lists then we could just fold with the arguments

```
0 :: Int                (+) :: Int → Int → Int
```

Is it possible to use these arguments in an algebra family fold over sequences? The answer is yes — we can use these arguments to build an algebra family that will perform the summation. Let us look at some sequence values and think about how we could apply 0 and (+) (we will use (+) in its prefix form so as to make clear the generalization to arbitrary fold arguments.

(i) The simplest sequence is NilS, or the empty sequence, and we can certainly replace this with 0.

(ii) For a sequence with one element, say (OneS 6 NilS), if we replace NilS by 0 then we can also replace OneS by (+). The value then becomes ((+) 6 0) which evaluates to 6 as required.

(iii) For a sequence of two elements, say (ZeroS (OneS (5,7) NilS)), we can no longer simply replace OneS with (+) because its first argument is now a *pair* of integers; however, we can construct a new function that operates on pairs:

```
\ (x,y) z → (+) x ((+) y z) :: Pair Int → Int → Int
```

by adding the values in the correct order. What about the ZeroS constructor? A ZeroS constructor contributes no elements to a sequence, so there is nothing to add, and we can replace it with id so that it simply passes the computed value through. Thus we get:

```
    (id ((+) 5 ((+) 7 0)))
=
    ((+) 5 ((+) 7 0))
=
    ((+) 5 7)
=
    12
```

To summarize, we replace all NilS constructors by 0 and all ZeroS constructors by id. We can replace the Int instance of OneS by (+), but for further instances, (P Int), (P (P Int)), and so on, we have to generate new functions that add pairs, and then pairs of pairs, and so on. This information is enough to allow us to generate a suitable algebra family using our general purpose combinator generateSeqAlgFam':

```
sumSeqAlgFam' :: SeqAlgFam' Int Int
sumSeqAlgFam' = generateSeqAlgFam'
                    0 id
                    id id
                    (+) (\ h → \ (x,y) z → h x (h y z))
```

then we can define

```
sumSeq :: Seq Int → Int
sumSeq = foldSeqAlgFam' sumSeqAlgFam'
```

and this will sum any (finite) sequence of integers.

Of course, there was nothing special about our choice of 0 and (+) as the arguments to the list fold — the above process would work for any arguments:

```
e :: b          f :: a → b → b
```

and in this way we can define a general combinator that simulates list folds on sequences:

```
listFoldSeq :: b → (a → b → b) → (Seq a → b)
listFoldSeq e f = foldSeqAlgFam' fam
  where fam = generateSeqAlgFam'
                e id
                id id
                f (\ h → \ (x,y) z → h x (h y z))
```

Then

```
sumSeq = listFoldSeq 0 (+)
```

In particular, this gives us an elegant way to collapse a sequence into a list — simply fold with the list constructors:

```
seqToList :: Seq a → List a
seqToList = listFoldSeq NilL ConsL
```

4.8. Chapter summary

The limited usefulness of functor category folds was a result of the replacement functions having to be parametrically polymorphic. In this chapter we loosened the restrictions by considering explicitly defined replacement *families* of functions whose members need not satisfy naturality conditions. For each non-uniform data-type, an infinite data-type is defined to capture families of replacement functions of suitable type. We call these *algebra families*. It is then straightforward to define fold combinators that replace constructor instances with members taken from a given family, and these are called *algebra family folds*.

The typing of the algebra families is fairly general and allows the construction of families for many useful operations, including non-parametric polymorphic operations that could not be expressed as functor category catamorphisms. In fact, the full generality of algebra family typing is seldom required and it can be profitable to define families with more specialized types to make the construction of such families simpler.

The construction of particular algebra families is by no means a trivial task, with an algebra family type typically being an infinite non-uniform type itself. However, by judiciously specializing the algebra family type and providing combinators to aid in the construction of algebra families, much of the intricacy can be hidden from the programmer who need supply only a few simple arguments from which the combinators build the desired algebra.

Chapter 5. Categorical semantics for algebra family folds

At the beginning of chapter 4 we stated that we were moving away from the categorical framework of initial algebras that underlies the traditional theory for folds for regular types and the functor category folds of chapter 3, and were instead taking a more pragmatic viewpoint by following the slogan that "folds replace constructors". Now we attempt to place the process we have described in a categorical framework similar to that of folds for regular types.

Suppose we traverse a data-value x, replacing constructor instances using a fold to get a resulting expression y.

$$x \mapsto y$$

If we can then traverse the resulting expression, replacing the first set of *replacement* functions with some new replacements to get a new result z,

$$x \mapsto y \mapsto z$$

then it would seem reasonable to combine the two traversals into a single fold that omits the intermediate step and directly transforms x into z:

$$x \mapsto z$$

This is the idea behind the *fusion* rule for catamorphisms. But, intuitively at least, such an idea seems reasonable for algebra family folds as well. Fusion rules are often the result of *initiality* in some category — catamorphism fusion derives from the fact that the data-types and their constructors form initial objects in categories of algebras, which is why they are called initial algebras. If we are to justify such a fusion rule for algebra family folds then we must find from which initial object in which category the fusion rule is derived. This is the focus of the remainder of this chapter, and leads us to introduce the notion of *initial algebra families* for which our algebra family folds turn out to be catamorphisms.

In identifying the constructor instances that may occur in a data-value of a particular type, we discovered that the instances correspond to finite strings of the modifying functors. To then fold over that data-value, we need an algebra family that contains replacements corresponding to each string. In the rest of this chapter, we model these algebra families as though they were indexed sets — the members of a family being indexed by the corresponding strings of modifying functors. These indexed families can be conveniently treated within our existing framework of functor categories, and many of the results of chapter 3 translate more-or-less directly to our indexed families, thus we achieve considerable economy of effort by removing the need to duplicate results.

Obviously there is some discrepancy between indexed families modelled as indexed sets and the algebra family types we define in Haskell which are infinite tree structures. The latter can be viewed as an implementation of the former.

5.1. Set-indexed families of objects and arrows

We begin by formalizing what we mean by "indexed families", and show that they fit naturally into functor categories. We wish to talk about families of objects and arrows of our base category $\mathbb{C}$ indexed by sets. That is, for an indexing set I, we would like an I-indexed family of objects of $\mathbb{C}$ to contain a single $\mathbb{C}$-object for each i in I. We will write such families as $\left\{X_i\right\}_{i\in I}$. Between two I-indexed families of $\mathbb{C}$-objects we can define I-indexed families of $\mathbb{C}$-arrows $\left\{f_i : X_i \to Y_i\right\}_{i\in I}$ where f_i is a $\mathbb{C}$-arrow from the i-th member of $\left\{X_i\right\}_{i\in I}$ to the i-th member of $\left\{Y_i\right\}_{i\in I}$. It turns out that these notions of families of objects and arrows can be captured as functors and natural transformations respectively for an appropriate choice of categories.

Any set I can be considered as a *discrete category* where the objects are the elements of I and the only arrows are the identity arrows. Then consider a functor $F : I \to \mathbb{C}$ from the discrete category to some other category $\mathbb{C}$. This functor defines a mapping from the elements of I to the objects of $\mathbb{C}$. The arrow part of the functor mapping is irrelevant because the only arrows in I are the identity arrows, and they must map to the corresponding identity arrows in $\mathbb{C}$. In effect, F describes an I-indexed family of $\mathbb{C}$-objects, which we could write as $\left\{F.i\right\}_{i\in I}$ in our notation for families. Conversely, each I-indexed family of $\mathbb{C}$-objects defines a functor $I \to \mathbb{C}$ by simply mapping elements of I to the corresponding object in the family. Now suppose we have two I-indexed families considered as functors $X, Y : I \to \mathbb{C}$. Then a natural transformation between them, $\alpha : X \overset{\cdot}{\longrightarrow} Y$ is a family of $\mathbb{C}$-arrows, $\left\{\alpha_i : X.i \to Y.i\right\}_{i\in I}$, indexed by elements of I and satisfying the naturality requirement. However, because the only arrows in I are the identity arrows, the naturality requirement is trivially satisfied by any I-indexed family of $\mathbb{C}$-arrows. Thus

- I-indexed families of $\mathbb{C}$-objects can be represented by functors $I \to \mathbb{C}$;
- I-indexed families of $\mathbb{C}$-arrows can be represented by natural transformations between functors $I \to \mathbb{C}$.

We can compose two natural transformations, and this corresponds to pointwise composing all the members of the two families that the natural transformations represent. That is, if we have three families of $\mathbb{C}$-objects, $\left\{X_i\right\}_{i\in I}$, $\left\{Y_i\right\}_{i\in I}$ and $\left\{Z_i\right\}_{i\in I}$, and two families of arrows between them, $\left\{\alpha_i : X_i \to Y_i\right\}_{i\in I}$ and $\left\{\beta_i : Y_i \to Z_i\right\}_{i\in I}$, considered as natural transformations $\alpha : X \overset{\cdot}{\longrightarrow} Y$ and $\beta : Y \overset{\cdot}{\longrightarrow} Z$, then composition of α and β gives us the family:

$$\left\{\beta_i \circ \alpha_i : X_i \to Z_i\right\}_{i\in I}$$

With composition of natural transformations we arrive at the functor category $\mathbb{C}^I$ whose objects are I-indexed families of $\mathbb{C}$-objects and arrows are I-indexed families of $\mathbb{C}$-arrows.

5.2. Finding the indexing set

We will model our algebra families as being indexed by some set I. As we have seen, the members of our families are in correspondance with finite strings of the modifying functors for the particular data-type we are dealing with, so we choose the indexing set to be the set of all such strings. For nests, which have only a single modifying functor P, the indexing set is simply

$$\{P\}^* = \{Id, P, PP, PPP, \dots\}$$

where we write Id for the empty string. It is important to realize that we are just dealing with strings of the *names* of the modifying functors, we are not yet actually *composing* the strings of functors to create new functors. For data-types with two modifying functors, say P and Q, the indexing set is more interesting:

$$\{P, Q\}^* = \{Id, P, Q, PP, PQ, QP, QQ, PPP, \dots\}$$

and it can be seen that these correspond to the constructor instances in figure 4.1. We can define such indexing sets for data-types with any finite number of modifying functors.

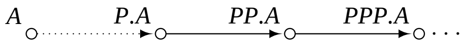
In what follows, once we have chosen a particular data-type, the indexing set remains fixed and we can define all our folds on that data-type using the same indexing set. In other words, once the data-type is chosen, we can perform all our work inside the functor category $\mathbb{C}^I$.

5.3. Next functors

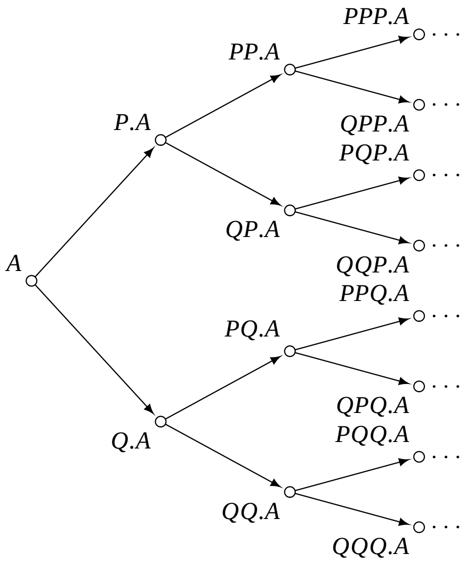
Recall the algebra family fold we defined over nests:

```
foldNestAlgFam :: NestAlgFam r s a b → Nest a → r b
foldNestAlgFam fam NilN = (nilnreplace fam)
foldNestAlgFam fam (ConsN x xs) =
          (consnreplace fam) x (foldNestAlgFam (nextP fam) xs)
```

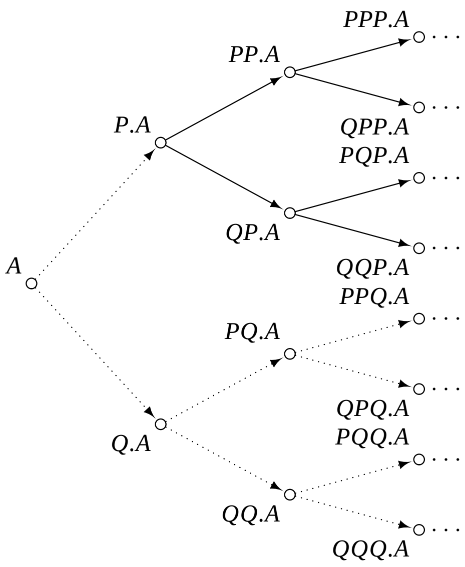
Notice that the algebra family we pass to the recursive call is not the same as the original algebra family — it has the nextP selector function applied to it. This has the effect of discarding the current replacements, as they are no longer needed, and moving the start of the algebra family to the next replacements. Visually, if we picture the algebra family as a list (a unary tree):



where the nodes contain the replacements for the constructors at the instances shown, then the dotted node is discarded and the black portion of the diagram becomes the new algebra family. If our data-type has two modifying functors, say P and Q, then we would have a binary tree:



In this case we would need *two* next functions — one to discard the lower subtree and move the root of the algebra family to the $P.A$ instance:



And another to do the same, but discarding the upper subtree and moving to the lower subtree.

In order to be able to express our recursion pattern categorically, we need to somehow capture the operation of these next functions on our indexed families. Fortunately this can easily be achieved by *re-indexing* our indexed families. Let us begin by briefly explaining re-indexing. Suppose we have a family $X = \left\{ X_I \right\}_{i \in I}$ indexed by some set I, and a function $f : J \to I$ that maps elements of some other set J to elements of I. We can then construct a new family $\left\{ X_{f(j)} \right\}_{j \in J}$ that is indexed by the set J. How does this work categorically? A function $f : J \to I$ between sets is trivially a functor between the sets J and I considered as *discrete* categories. Because our family X is also a functor, this time from I to $\mathbb{C}$, we can simply compose the two functors

$$J \xrightarrow{f} I \xrightarrow{X} \mathbb{C}$$

to get a new functor $Xf : J \to \mathbb{C}$, which is a J-indexed family of $\mathbb{C}$-objects. If we examine the j-th element of Xf for some j in J, then we see

$Xf.j$

$=$ { composition of functors }

$X.(f.j)$

$=$ { writing in usual function notation }

$X_{f(j)}$

so we are performing exactly the re-indexing described above.

This re-indexing works not only for families of $\mathbb{C}$-objects, but also for families of $\mathbb{C}$-arrows. Let X and Y be two families of $\mathbb{C}$-objects, and $\alpha = \left\{ \alpha_I : X_I \to Y_I \right\}_{i \in I}$ a family of $\mathbb{C}$-arrows between them. We know that α is a natural transformation between X and Y, and we can compose functors and natural transformations, so we get a new natural transformation $\alpha f : Xf \to Yf$. For any j in J, the j-th component of αf is

$(\alpha f)_j$

$=$ { composition of functors and natural transformations }

$\alpha_{f.j}$

$=$ { writing in usual function notation }

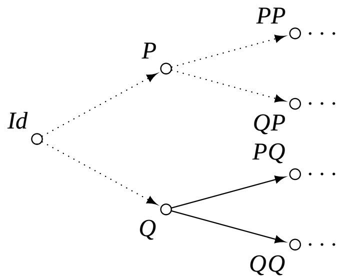
$\alpha_{f(j)}$

So we can re-index both families of objects and families of arrows, and this allows us to construct *re-indexing functors*:

> **Definition 5.1 (re-indexing functor):** For a category $\mathbb{C}$, two indexing sets I and J, and a function $f : J \to I$ between them, we define the *re-indexing functor* to be
>
> $$\begin{tikzcd}[row sep=small] (\circ f) : \mathbb{C}^I & \longrightarrow & \mathbb{C}^J \\ X \arrow[dd, "\alpha"'] & & Xf \arrow[dd, "\alpha f"] \\ & \mapsto & \\ Y & & Yf \end{tikzcd}$$

The action of this functor is simply to precompose the function f with a functor or natural transformation.

How do we use these re-indexing functors? Suppose we have a family of $\mathbb{C}$-objects indexed by the set $\{P, Q\}^*$, and we wish to discard the P branch and move the root down the Q branch:



We use the function that appends Q to the front of all the string in the indexing set:

$$(Q{+\!\!+}) : \begin{array}{ccc} \{P, Q\}^* & \rightarrow & \{P, Q\}^* \\ x & \mapsto & Qx \end{array}$$

where $+\!\!+$ is the string concatenation operator. Remember, at this level we are not dealing with the *functors* P and Q, we are just using strings formed out of their *names* as a convenient form of indexing. Then we get the re-indexing functor:

$$(\circ(Q{+\!\!+})) : \mathbb{C}^{\{P,Q\}^*} \to \mathbb{C}^{\{P,Q\}^*}$$

that transforms indexed families. We can see the effect of applying $(\circ(Q{+\!\!+}))$ to some indexed family $X = \{X_i\}_{i \in \{P,Q\}^*}$ of $\mathbb{C}$-objects — for any i in $\{P, Q\}^*$, the i-th component of $((\circ(Q{+\!\!+})).X)$ is equal to the Qi-th component of X:

$$\begin{array}{ll}
& ((\circ(Q{+\!\!+})).X).i \\
= & \qquad \{ \text{definition of } (\circ(Q{+\!\!+})) \} \\
& X(Q{+\!\!+}).i \\
= & \qquad \{ \text{composition of functors} \} \\
& X.((Q{+\!\!+}).i) \\
= & \qquad \{ \text{definition of } (Q{+\!\!+}) \} \\
& X.(Qi)
\end{array}$$

The effect is the same for indexed families of $\mathbb{C}$-arrows.

This functor performs precisely the re-indexing we require, so we name it nextQ:

$$\mathsf{nextQ} = (\circ(Q\mathbin{+\!\!+})) : \mathbb{C}^{\{P,Q\}^*} \to \mathbb{C}^{\{P,Q\}^*}$$

We can similarly define "next" functors for the modifying functors of each recursive call, so we have also

$$\mathsf{nextP} = (\circ(P\mathbin{+\!\!+})) : \mathbb{C}^{\{P,Q\}^*} \to \mathbb{C}^{\{P,Q\}^*}$$

5.4. Initial algebra families and catamorphisms

We will again use nests as our motivating example:

```
type P a = (a,a)
data Nest a = NilN
            | ConsN a (Nest (P a))
```

Fix some type A, then we can consider the nest constructors at the instances A, $P.A$, $P.P.A$, and so on, as an indexed family in the following way. For each string Ω in $\{P\}^*$, the corresponding constructor instance has type:

$$\begin{tikzcd} \mathbb{1} + \Omega.A \times Nest.P.\Omega.A \arrow[d, "\mathsf{NilN}_{\Omega.A} \triangledown \mathsf{ConsN}_{\Omega.A}"] \\ Nest.\Omega.A \end{tikzcd}$$

So let $\{P\}^*$ be our indexing set I, and define the family $\gamma \overset{def}{=} \{\mathsf{NilN}_{\Omega.A} \triangledown \mathsf{ConsN}_{\Omega.A}\}_{\Omega\in I}$, which is a family of arrows from $\{\mathbb{1} + \Omega.A \times Nest.P.\Omega.A\}_{\Omega\in I}$ to $\{Nest.\Omega.A\}_{\Omega\in I}$. All the families we will deal with in the following section will be indexed by the same set $I = \{P\}^*$ so we will omit the "$\Omega \in I$" subscript on families to avoid clutter in our calculations.

The constructors form an algebra for a functor $F : \mathbb{C}^I \to \mathbb{C}^I$. We can easily calculate F by observing that the carrier of the algebra is $\{\, Nest.\Omega.A \,\}$ — the target family of γ. Then we know that the source family of γ must be F applied to the carrier, so we "factor" the carrier out of the source family:

$$\begin{array}{ll}
 & \{\, \mathbb{1} + \Omega.A \times Nest.P.\Omega.A \,\} \\
= & \qquad \{\text{ pointwise sums and products of families }\} \\
 & \{\, \mathbb{1} \,\} \mathbin{\dot{+}} \{\, \Omega.A \,\} \mathbin{\dot{\times}} \{\, Nest.P.\Omega.A \,\} \\
= & \qquad \{\text{ definition of } nextP \text{ }\} \\
 & \{\, \mathbb{1} \,\} \mathbin{\dot{+}} \{\, \Omega.A \,\} \mathbin{\dot{\times}} nextP.\{\, Nest.\Omega.A \,\}
\end{array}$$

This leads us to define F as:

$$\begin{tikzcd}
F : \mathbb{C}^I \arrow[r, phantom, "\longrightarrow"] & \mathbb{C}^I \\[-2em]
X \arrow[d, "\alpha"'] \arrow[r, phantom, "\mapsto"] & \{\, \mathbb{1} \,\} \mathbin{\dot{+}} \{\, \Omega.A \,\} \mathbin{\dot{\times}} nextP.X \arrow[d, "Id \mathbin{\dot{+}} Id \mathbin{\dot{\times}} nextP.\alpha"] \\
Y & \{\, \mathbb{1} \,\} \mathbin{\dot{+}} \{\, \Omega.A \,\} \mathbin{\dot{\times}} nextP.Y
\end{tikzcd}$$

We could write this functor more compactly using double lifting as:

$$F = \underline{\{\, \mathbb{1} \,\}} \mathbin{\ddot{+}} \underline{\{\, \Omega.A \,\}} \mathbin{\ddot{\times}} nextP$$

Then we have that F applied to the target family of γ equals the source family of γ:

$$F.\{\, Nest.\Omega.A \,\} = \{\, \mathbb{1} + \Omega.A \times Nest.P.\Omega.A \,\}$$

as required. So γ is an F-algebra.

We will defer proving the existence of an initial algebra for F to section 5.7. Let us assume that γ is an initial algebra for F, and to indicate this, we will henceforth call it in^F:

$$in^F \overset{def}{=} \gamma = \left\{ \mathsf{NilN}_{\Omega.A} \triangledown \mathsf{ConsN}_{\Omega.A} \right\}_{\Omega \in I}$$

and call the carrier μF:

$$\mu F \overset{def}{=} \{\, Nest.\Omega.A \,\}$$

With initial algebras come catamorphisms — the unique algebra homomorphisms to other algebras — and we now show that these catamorphisms formalize our notion of algebra family folds.

For any F-algebra $\alpha : F.X \to X$, we get a unique F-algebra homomorphism, $\llparenthesis\, \alpha \,\rrparenthesis$, which makes the usual catamorphism diagram commute:

$$\begin{tikzcd}
F.\mu F \arrow[r, "F.\llparenthesis\, \alpha \,\rrparenthesis"] \arrow[d, "in^F"'] & F.X \arrow[d, "\alpha"] \\
\mu F \arrow[r, dotted, "\llparenthesis\, \alpha \,\rrparenthesis"'] & X
\end{tikzcd}$$

and because in^F is an isomorphism, we get the usual recursive definition for $\llparenthesis\, \alpha \,\rrparenthesis$:

$$\llparenthesis\, \alpha \,\rrparenthesis = \alpha \circ F.\llparenthesis\, \alpha \,\rrparenthesis \circ \left(in^F \right)^{-1}$$

Expanding F in the diagram, and letting α be $(\lambda \mathbin{\dot{\triangledown}} \rho)$ for some families λ and ρ, we have the diagram:

$$\begin{tikzcd}
\{\, \mathbb{1} \,\} \dotplus \{\, \Omega.A \,\} \mathbin{\dot{\times}} nextP.\mu F \arrow[r, "Id \dotplus Id \mathbin{\dot{\times}} nextP.\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis"] \arrow[d, "\left\{ \mathsf{NilN}_{\Omega.A} \right\} \mathbin{\dot{\triangledown}} \left\{ \mathsf{ConsN}_{\Omega.A} \right\}"'] & \{\, \mathbb{1} \,\} \dotplus \{\, \Omega.A \,\} \mathbin{\dot{\times}} nextP.X \arrow[d, "\lambda \mathbin{\dot{\triangledown}} \rho"] \\
\mu F \arrow[r, dotted, "\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis"'] & X
\end{tikzcd}$$

which is equivalent to the two equations:

$$\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis \circ \left\{ \mathsf{NilN}_{\Omega.A} \right\} = \lambda \circ Id = \lambda \qquad (5.4.1)$$

and

$$\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis \circ \left\{ \mathsf{ConsN}_{\Omega.A} \right\} = \rho \circ \left(\mathit{Id} \,\dot{\times}\, \mathit{nextP}.\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis \right) \tag{5.4.2}$$

But remember that all the arrows here, including $\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis$, are *families* of arrows of our base category. What we really want is not $\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis$, but the first member of that family:

$$\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}} : \mathit{Nest}.A \to X_{\mathit{Id}}$$

which is now a single arrow in our base category that takes nests of A's to values of the type that is the first member of X. Let us look at the behaviour of $\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}}$ when applied to the constructors NilN_A and ConsN_A. First we apply it to the NilN_A constructor:

$$\begin{aligned}
& \llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}} \circ \mathsf{NilN}_A \\
= \quad & \{ \text{ member selection } \} \\
& \llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}} \circ \left\{ \mathsf{NilN}_{\Omega.A} \right\}_{\mathit{Id}} \\
= \quad & \{ \text{ pointwise composition } \} \\
& \left(\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis \circ \left\{ \mathsf{NilN}_{\Omega.A} \right\} \right)_{\mathit{Id}} \\
= \quad & \{ \text{ equation 5.4.1 } \} \\
& \lambda_{\mathit{Id}}
\end{aligned}$$

This produces the first member of the λ family, $\lambda_{\mathit{Id}} : \mathbb{1} \to X_{\mathit{Id}}$. The recursion comes into play when $\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis$ is applied to the ConsN_A constructor:

$$\begin{aligned}
& \llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}} \circ \mathsf{ConsN}_A \\
= \quad & \{ \text{ member selection } \} \\
& \llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis_{\mathit{Id}} \circ \left\{ \mathsf{ConsN}_{\Omega.A} \right\}_{\mathit{Id}} \\
= \quad & \{ \text{ pointwise composition } \} \\
& \left(\llparenthesis\, \lambda \,\dot{\triangledown}\, \rho \,\rrparenthesis \circ \left\{ \mathsf{ConsN}_{\Omega.A} \right\} \right)_{\mathit{Id}} \\
= \quad & \{ \text{ equation 5.4.2 } \}
\end{aligned}$$

$$\left(\rho \circ \left(Id_{\{\Omega.A\}} \mathbin{\dot{\times}} \left(nextP.\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis\right)\right)\right)_{Id}$$

$=$ { pointwise composition }

$$\rho_{Id} \circ \left(Id_A \times \left(nextP.\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis\right)_{Id}\right)$$

$=$ { definition of *nextP* }

$$\rho_{Id} \circ \left(Id_A \times \llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis_P\right)$$

In the same way, one can verify that for any Ω in $\{P\}^*$,

$$\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis_\Omega \circ \mathsf{NilN}_{\Omega.A} = \lambda_\Omega$$

and

$$\llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis_\Omega \circ \mathsf{ConsN}_{\Omega.A} = \rho_\Omega \circ \left(Id_{\Omega.A} \times \llparenthesis\, \lambda \mathbin{\dot{\triangledown}} \rho \,\rrparenthesis_{P\Omega.A}\right)$$

So $\mathsf{NilN}_{\Omega.A}$ and $\mathsf{ConsN}_{\Omega.A}$ are replaced by the corresponding members of the families λ and ρ. Clearly then, this is the same behaviour as the foldNest combinator we defined previously:

```
foldNestAlgFam :: NestAlgFam r s a b → Nest a → r b
foldNestAlgFam fam NilN = (nilnreplace fam)
foldNestAlgFam fam (ConsN x xs) =
          (consnreplace fam) x (foldNestAlgFam (next fam) xs)
```

where the values of nilnreplace correspond to members of the λ family, and the values of consnreplace correspond to members of the ρ family.

We should say something about the *types* of the algebra families specified in the foldNest combinator. The semantics we have described will accept *any* family $\alpha : F.X \to X$ as the argument algebra — no restrictions are placed on the objects X_Ω in the carrier family. However, in Haskell the type system prevents us from specifying families with such arbitrary types, and so we are forced to define the more restricted type of algebra families:

```
data NestAlgFam r s a b =
        MkNestAlgFam {
          nilnreplace :: r b ,
          consnreplace :: a → r (s b) → r b ,
          next :: NestAlgFam r s (P a) (s b)
        }
```

However, it is easy to check that every NestAlgFam is an F-algebra. The carrier of a NestAlgFam with parameters R, S, A and B is a family

$$\left\{R.\Phi(\Omega).B\right\}_{\Omega\in I}$$

where Φ maps strings of P's to strings of S's of the same length. The simpler algebra families for nests, where R and S are assumed to be the identity:

```
data NestAlgFam' a b =
        MkNestAlgFam' {
          nilnreplace' :: b ,
          consnreplace' :: a → b → b ,
          next' :: NestAlgFam' (P a) b
        }
```

correspond to the simpler carrier family:

$$\{B\}_{\Omega\in I}$$

in which every member of the family is just B. Of course, every simplified nest algebra family is also an F-algebra.

5.5. Comparison with functor category catamorphisms

In chapter 3 we looked at catamorphisms for non-uniform types in the endofunctor category $\mathbb{C}^{\mathbb{C}}$. The arrows in $\mathbb{C}^{\mathbb{C}}$ are families of $\mathbb{C}$-arrows such that

(i) the families are indexed by the objects of $\mathbb{C}$;

(ii) the members of a family must satisfy the naturality requirement.

These two factors contributed to the restricted usefulness of functor category catamorphisms. One way of viewing our move to algebra families is as the loosening of these restrictions:

(i) our families are no longer indexed by *all* the objects in our base category, instead we have an indexing scheme that includes only the *possible* constructor instances;

(ii) by working in a functor category $\mathbb{C}^I$ where our source category is a *discrete category*, the naturality requirement on families is trivially satisfied by all families.

However, by introducing our own indexing mechanism instead of using that provided by natural transformations in $\mathbb{C}^{\mathbb{C}}$, the treatment becomes more complicated.

One simplification that occurs from working in $\mathbb{C}^I$ rather than $\mathbb{C}^{\mathbb{C}}$ is that we need no longer worry about the *size* of our categories when forming functor categories — our discrete category I will always be a set, and therefore small.

5.6. Transformation laws for algebra family folds

We have shown that algebra family folds correspond to catamorphisms from some initial algebra family. From the initiality of this algebra family we can derive versions of the usual catamorphism transformation laws for algebra family folds. The first of these says that if we fold with the initial algebra family itself as the argument family then the resulting catamorphism is the identity:

$$(\!|\, in^F \,|\!) = Id_{\mu F}$$

This makes sense intuitively because if we replace each constructor instance by itself then we don't change the overall value.

The most useful law is fusion — given two algebra families, $\alpha : F.X \to X$ and $\beta : F.Y \to Y$, and an algebra family homomorphism $h : X \to Y$ between them, then the following diagram commutes:

$$\begin{tikzcd}
F.\mu F \arrow[r, "F.\llparenthesis \alpha \rrparenthesis"] \arrow[d, "in^F"'] & F.X \arrow[r, "F.h"] \arrow[d, "\alpha"] & F.Y \arrow[d, "\beta"] \\
\mu F \arrow[r, dotted, "\llparenthesis \alpha \rrparenthesis"'] & X \arrow[r, "h"'] & Y
\end{tikzcd}$$

and by uniqueness of catamorphisms,

$$h \circ \llparenthesis \alpha \rrparenthesis = \llparenthesis \beta \rrparenthesis$$

Let us look more closely at the condition needed to allow fusion to be applied — we must have an algebra family homomorphism:

$$\begin{tikzcd}
F.X \arrow[r, "F.h"] \arrow[d, "\alpha"] & F.Y \arrow[d, "\beta"] \\
X \arrow[r, "h"'] & Y
\end{tikzcd} \tag{5.6.1}$$

But h must be a *family* of arrows satisfying diagram 5.6.1. For a particular i in I, $h_i : X_i \to Y_i$ must satisfy:

$$\begin{tikzcd}
(F.X)_i \arrow[r, "(F.h)_i"] \arrow[d, "\alpha_i"] & (F.Y)_i \arrow[d, "\beta_i"] \\
X_i \arrow[r, "h_i"'] & Y_i
\end{tikzcd}$$

So diagram 5.6.1 really captures infinitely many equations — one for each h_i. Furthermore, for a non-uniform data-type, the functor will contain one or more "next" functors, say $nextM_1, \dots, nextM_m$. Then the arrow $(F.h)_i$ will depend on the arrows $h_{M_1 i}, \dots, h_{M_m i}$. Therefore the equation for a particular h_i cannot be solved in isolation, as it involves other arrows in the family h. This tells us that establishing that a particular family $h : X \to Y$ is an algebra family homomorphism will not, in general, be straightforward, and will normally require additional knowledge about the construction of the algebra families α and β and relationships between the members of those families.

We now give an example of a situation where it is possible to simplify the solution of equation 5.6.1 by making certain assumptions. In order to be able to use the catamorphism fusion law for nests, we need a homomorphism between two algebra families:

$$\begin{tikzcd}
\{\mathbb{1}\} \dot{+} \{\mathit{\Omega}.A\} \dot{\times} \{R.S.\mathit{\Phi}(\mathit{\Omega}).B\} \arrow[r, "Id \dot{+} Id \dot{\times} \mathrm{nextP}.h"] \arrow[d, "\lambda \dot{\triangledown} \rho"] & \{\mathbb{1}\} \dot{+} \{\mathit{\Omega}.A\} \dot{\times} \{R'.S'.\mathit{\Phi}'(\mathit{\Omega}).B'\} \arrow[d, "\alpha \dot{\triangledown} \beta"] \\
\{R.\mathit{\Phi}(\mathit{\Omega}).B\} \arrow[r, "h"] & \{R'.\mathit{\Phi}'(\mathit{\Omega}).B'\}
\end{tikzcd}$$

where the sets are implicitly indexed by $\mathit{\Omega} \in \{P\}^*$ and $\mathit{\Phi}$ and $\mathit{\Phi}'$ map strings of "P"s to strings of "S"s and "S'"s respectively. The calculations become much simpler if we assume that the two algebra families are simplified algebra families where R, S, R' and S' are all the identity. Then the above diagram becomes

$$\begin{tikzcd}
\{\mathbb{1}\} \dot{+} \{\Omega.A\} \dot{\times} \{B\} \arrow[r, "Id \dot{+} Id \dot{\times} \mathsf{nextP}.h"] \arrow[d, "\lambda \dot{\triangledown} \rho"] & \{\mathbb{1}\} \dot{+} \{\Omega.A\} \dot{\times} \{B'\} \arrow[d, "\alpha \dot{\triangledown} \beta"] \\
\{B\} \arrow[r, "h"] & \{B'\}
\end{tikzcd}$$

Notice that the members of the family h now all have the same type:

$$\forall \Omega \in \{P\}^*, h_\Omega : B \to B'$$

and so we can simplify matters further by assuming that all the members of h are the same arrow. Let $h' : B \to B'$ be a single arrow in $\mathbb{C}$, then define

$$h \overset{def}{=} \{h'\}$$

Note that all nextP does is re-index the members of h, we have that

$$\mathsf{nextP}.h = h$$

because all the members are the same. Thus the above diagram becomes the equation

$$h \circ (\lambda \dot{\triangledown} \rho) = (\alpha \dot{\triangledown} \beta) \circ (Id \dot{+} Id \dot{\times} h)$$

But we are still working with infinite families, so the equation really corresponds to infinitely many equations about arrows in $\mathbb{C}$. To simplify matters further, we can make some assumptions about the way in which the families λ, ρ, α and β are constructed. Recall in section 4.5 we defined a combinator generateNestAlgFam' that constructed simplified nest algebra families from two seeds

$$n : \mathbb{1} \to B \qquad\qquad c : A \times B \to B$$

and a function that collapsed P structures:

$$\mathsf{collapseP} : P.A \to A$$

Then the two families nilnreplace and consnreplace are generated inductively from

$$\mathsf{nilnreplace}_{Id} = n$$

$$\forall \Omega \in \{P\}^*, \mathsf{nilnreplace}_{P\Omega} = \mathsf{nilnreplace}_{\Omega}$$

and

$$\mathsf{consnreplace}_{Id} = c$$

$$\forall \Omega \in \{P\}^*, \mathsf{consnreplace}_{P\Omega} = \mathsf{consnreplace}_{\Omega} \circ (\Omega.\mathsf{collapseP} \times Id)$$

If we assume that both our algebra families are generated using this combinator with the same collapseP, that is, we are given λ_{Id}, ρ_{Id}, α_{Id}, β_{Id} and collapseP and we know that for any Ω in $\{P\}^*$,

$$\lambda_{P\Omega} = \lambda_{\Omega} \qquad \rho_{P\Omega} = \rho_{\Omega} \circ (\Omega.\mathsf{collapseP} \times Id) \tag{5.6.2}$$

and

$$\alpha_{P\Omega} = \alpha_{\Omega} \qquad \beta_{P\Omega} = \beta_{\Omega} \circ (\Omega.\mathsf{collapseP} \times Id) \tag{5.6.3}$$

then we can try and prove the equation of families

$$h \circ (\lambda \mathbin{\dot{\triangledown}} \rho) = (\alpha \mathbin{\dot{\triangledown}} \beta) \circ (Id \mathbin{\dot{+}} Id \mathbin{\dot{\times}} h)$$

by induction. That is, if we can prove the single equation for the base case:

$$h' \circ (\lambda_{Id} \triangledown \rho_{Id}) = (\alpha_{Id} \triangledown \beta_{Id}) \circ (Id + Id \times h')$$

then the rest of the equations follow by induction. To see this we must prove the induction step: for any Ω in $\{P\}^*$:

$$\begin{aligned}
& h' \circ (\lambda_{\Omega} \triangledown \rho_{\Omega}) = (\alpha_{\Omega} \triangledown \beta_{\Omega}) \circ (Id + Id \times h') \\
\equiv \quad & \{ \text{split into two equations} \} \\
& h' \circ \lambda_{\Omega} = \alpha_{\Omega} \wedge h' \circ \rho_{\Omega} = \beta_{\Omega} \circ (Id \times h')
\end{aligned}$$

$\Rightarrow$ { induction step — proved below }

$$h' \circ \lambda_{P\Omega} = \alpha_{P\Omega} \wedge h' \circ \rho_{P\Omega} = \beta_{P\Omega} \circ (Id \times h')$$

$\equiv$ { join into a single equation }

$$h' \circ (\lambda_{P\Omega} \triangledown \rho_{P\Omega}) = (\alpha_{P\Omega} \triangledown \beta_{P\Omega}) \circ (Id + Id \times h')$$

For λ and α the induction step is easy:

$$h' \circ \lambda_{P\Omega}$$

$=$ { definition of λ (5.6.2) }

$$h' \circ \lambda_{\Omega}$$

$=$ { assumption }

$$\alpha_{\Omega}$$

$=$ { definition of α (5.6.3) }

$$\alpha_{P\Omega}$$

The calculation for ρ and β is slightly longer:

$$h' \circ \rho_{P\Omega}$$

$=$ { definition of ρ (5.6.2) }

$$h' \circ \rho_{\Omega} \circ (\Omega.\mathsf{collapseP} \times Id)$$

$=$ { assumption }

$$\beta_{\Omega} \circ (Id \times h') \circ (\Omega.\mathsf{collapseP} \times Id)$$

$=$ { identities and products }

$$\beta_{\Omega} \circ (\Omega.\mathsf{collapseP} \times h')$$

$=$ { identities and products }

$$\beta_{\Omega} \circ (\Omega.\mathsf{collapseP} \times Id) \circ (Id \times h')$$

$=$ { definition of β (5.6.3) }

$$\beta_{P\Omega} \circ (Id \times h')$$

Thus we have reduced (under some fairly strong assumptions) the problem of showing that a family of $\mathbb{C}$-arrows is an algebra homomorphism to proving a single equation of $\mathbb{C}$-arrows — the base case of the induction:

$$h' \circ (\lambda_{Id} \triangledown \rho_{Id}) = (\alpha_{Id} \triangledown \beta_{Id}) \circ (Id + Id \times h')$$

The assumptions we made were:

(i) $(\lambda \mathbin{\dot{\triangledown}} \rho)$ and $(\alpha \mathbin{\dot{\triangledown}} \beta)$ were simplified nest algebra families where the result and step functors were all the identity;

(ii) $(\lambda \mathbin{\dot{\triangledown}} \rho)$ and $(\alpha \mathbin{\dot{\triangledown}} \beta)$ were generated inductively using the generateNestAlgFam' pattern, from the seeds λ_{Id}, ρ_{Id}, α_{Id} and β_{Id}, and the same function collapseP;

(iii) the family h was a constant family with all members equalling some $h' : B \to B'$.

To see that this *is* actually a useful result, we give the following example of its use. If we take A to be the type $[String]$ of lists of strings (we use the Haskell notation for list types for convenience), B to be Int, the type of integers, and collapseP to be $+\!\!+$, the function that joins two lists into a single list:

$$+\!\!+ : P.[String] \to [String]$$

then we can construct an algebra family $(\lambda \mathbin{\dot{\triangledown}} \rho)$ that collapses a nest of lists of strings into a single list of strings by using generateNestAlgFam' with the seeds

$$\lambda_{Id} \overset{def}{=} [\,] : \mathbb{1} \to [String] \tag{5.6.4}$$

and

$$\rho_{Id} \overset{def}{=} +\!\!+ : [String] \times [String] \to [String] \tag{5.6.5}$$

and the collapseP function defined above. We can also define a second algebra family $(\alpha \mathbin{\dot{\triangledown}} \beta)$ that counts the number of strings in a nest of list of strings, again by using generateNestAlgFam', but this time with the seeds

$$\alpha_{Id} \overset{def}{=} \mathsf{zero} : \mathbb{1} \to Int \tag{5.6.6}$$

and

$$\beta_{Id} \overset{def}{=} [String] \times Int \xrightarrow{\mathsf{length} \times Id} Int \times Int \xrightarrow{\oplus} Int \tag{5.6.7}$$

where length counts the number of elements in a list, and $\oplus$ is addition on integers. It should be obvious that

$$\mathsf{length} \circ (\!|\, \lambda \dot{\triangledown} \rho \,|\!)_{Id} = (\!|\, \alpha \dot{\triangledown} \beta \,|\!)_{Id} \tag{5.6.8}$$

because if we collapse a nest into a list and then count the strings in the list then we should get the same result as just counting the strings in the nest. We wish to prove equation 5.6.8 using fusion. We need to show that $\left\{ \mathsf{length} \right\}$ is an algebra homomorphism from $(\lambda \dot{\triangledown} \rho)$ to $(\alpha \dot{\triangledown} \beta)$. We are not concerned with the implementation of length and ++, except that they must satisfy the following obvious conditions:

(i) the length of an empty list is zero:

$$\mathsf{length} \circ [\,] = \mathsf{zero} \tag{5.6.9}$$

(ii) the length of the concatenation of two lists is the sum of the lengths of the two lists:

$$\mathsf{length} \circ +\!\!+ = \oplus \circ (\mathsf{length} \times \mathsf{length}) \tag{5.6.10}$$

then all we have to do is prove the induction base case with

$$h' \overset{def}{=} \mathsf{length} \tag{5.6.11}$$

We calculate:

$$(\alpha_{Id} \triangledown \beta_{Id}) \circ (Id + Id \times h')$$

$=$ { definitions of α_{Id}, β_{Id} and h' (5.6.6, 5.6.7, 5.6.11) }

$$(\mathsf{zero} \triangledown \oplus \circ (\mathsf{length} \times Id)) \circ (Id + Id \times \mathsf{length})$$

$=$ { sum fusion, identities }

$\mathsf{zero} \triangledown \oplus \circ (\mathsf{length} \times \mathsf{length})$

$=$ { equations 5.6.9 and 5.6.10 }

$\mathsf{length} \circ [\,] \triangledown \mathsf{length} \circ +\!\!+$

$=$ { sum fusion }

$\mathsf{length} \circ ([\,] \triangledown +\!\!+)$

$=$ { definitions of h', λ_{Id}, ρ_{Id} (5.6.11, 5.6.4, 5.6.5) }

$h' \circ (\lambda_{Id} \triangledown \rho_{Id})$

Note that collapseP does not appear at all in the proof of the base case — it is dealt with automatically by the induction step.

We can easily code the above algebra families in Haskell — first we define collapseP:

```
collapseP :: (P [String]) → [String]
collapseP (MkP (xs, ys)) = xs ++ ys
```

then the families $(\lambda \dot{\triangledown} \rho)$ and $(\alpha \dot{\triangledown} \beta)$ are generated by:

```
lamRho :: NestAlgFam' [String] [String]
lamRho = generateNestAlgFam' lam0 rho0 collapseP
  where lam0 = []
        rho0 = (++)
```

and

```
alphaBeta :: NestAlgFam' [String] Int
alphaBeta = generateNestAlgFam' alpha0 beta0 collapseP
  where alpha0 = 0
        beta0 = \ xs y → (length xs) + y
```

If we define the convenient shorthand definitions:

```
collapseNest = foldNestAlgFam' lamRho
```

and

```
countNest = foldNestAlgFam' alphaBeta
```

then we have proved that

```
length . collapseNest = countNest
```

and we have eliminated the intermediate list structure.

In chapter 6 we will use the transformation laws to prove laws about maps over non-uniform data-types expressed as algebra family folds. Our use of fusion there to prove that maps preserve composition is an example of a situation where it *is* straightforward to establish an appropriate algebra family homomorphism.

5.7. Existence of initial algebra families

We now look at conditions under which we can prove that certain initial algebra families exist. As we did for functor categories in chapter 3, we will lift the standard results described in section 2.5 to apply to algebra families in the category $\mathbb{C}^I$. The fact that $\mathbb{C}^I$ is a functor category means that we can re-use many of the results we established in chapter 3 about completeness and continuity properties and about the lifting of functors.

Theorem 3.2 applies directly for $\mathbb{C}^I$ for any indexing set I, and the resulting corollaries lead immediately to a version of the generalized Kleene fixed-point theorem for $\mathbb{C}^I$:

> **Theorem 5.1:** If $\mathbb{C}$ is an ω-cocomplete category with an initial object, I is any indexing set and $F : \mathbb{C}^I \to \mathbb{C}^I$ is an ω-cocontinuous functor then F has an initial algebra.
>
> **Proof:** This is an instance of corollary 3.4.

Again, constant and identity functors are trivially ω-cocontinuous, and lemma 3.3

can be applied in $\mathbb{C}^I$ to show that functor lifting preserves continuity properties, in particular, for sums and products. In chapter 3 we also had to show ω-cocontinuity of precomposition functors such as $(\circ M)$. In $\mathbb{C}^I$, the base functors for initial algebras have "next" functors in place of the precomposition functors of chapter 3, but if we look more closely at the definition of a next functor, for example,

$$nextM \overset{def}{=} (\circ(M{+\!\!+}))$$

we see that they are, in fact, a kind of precomposition functor. Because $(M{+\!\!+})$ is *injective*, we can appeal to the following lemma to establish the ω-cocontinuity of $(\circ(M{+\!\!+}))$:

Lemma 5.1 (injective re-indexings are continuous and cocontinuous): Let I be a set and $f : I \to I$ an injective function. Consider I as a *discrete category* and f as an endofunctor on it. Then

$$(\circ f) : \mathbb{C}^I \longrightarrow \mathbb{C}^I \tag{5.7.1}$$

$$\begin{tikzcd} X \arrow[d, "\alpha"'] & \mapsto & Xf \arrow[d, "\alpha f"] \\ Y & & Yf \end{tikzcd}$$

is both continuous and cocontinuous.

Proof: In appendix A.5.

Corollary 5.1: If we have m modifying functors, $M_1, \ldots, M_m : \mathbb{C} \to \mathbb{C}$, and we define our indexing set I to be

$$I \overset{def}{=} \left\{ M_1, \ldots, M_m \right\}^*$$

then for $1 \leq i \leq m$, the functor $nextM_i$, defined to be

$$nextM_i \overset{def}{=} (\circ(M_i{+\!\!+})) : \mathbb{C}^I \to \mathbb{C}^I$$

is $\mathbb{J}$-continuous and $\mathbb{J}$-cocontinuous for any small category $\mathbb{J}$.

Proof: Each function $(M_i +\!\!+) : I \to I$ is injective, so we can appeal to lemma 5.1.

This is clearly a more useful result than we obtained in chapter 3, where we relied on our modifying functors having adjoints (corollary 3.5).

To summarize the results we have about ω-cocontinuity of functors $\mathbb{C}^I \to \mathbb{C}^I$, note that we can appeal to all the results of lemma 3.4 in chapter 3, and also lemma 5.1 for “next” functors.

Applying these results we see that our base functor for nest algebra families,

$$F = \underline{\{\,\mathbb{1}\,\}} \mathbin{\ddot{+}} \underline{\{\,\Omega.A\,\}} \mathbin{\ddot{\times}} nextP$$

is ω-cocontinuous provided that $\dot{+}$ and $\dot{\times}$ are ω-cocontinuous and our base category is ω-cocomplete.

5.7.1. Order-enriched results

The order-enriched results about the existence of initial algebras also lift straightforwardly to $\mathbb{C}^I$. Again, we can re-use much of the work we performed in chapter 3.

When we showed in lemma A.4 that functor categories can be made $\mathbf{O}\bot$-categories if their target categories are $\mathbf{O}\bot$-categories, we had to restrict our attention to strict subcategories of the target categories to achieve naturality. However, if the source category is *discrete*, so the only arrows are identity arrows, then the naturality requirement is satisfied trivially, and we don’t require strictness. Thus we get

Lemma 5.2 ($\mathbb{C}^I$ is an $\mathbf{O}\bot$-category if $\mathbb{C}$ is): If $\mathbb{C}$ is an $\mathbf{O}\bot$-category and I is any set considered as a discrete category then $\mathbb{C}^I$ is also an $\mathbf{O}\bot$-category.

Proof: The proof proceeds in much the same way as that of lemma A.4.

However, when proving $\perp_{F \to G}$ to be natural, the arrows $f : A \to B$ in I can only be identity arrows, and these trivially satisfy the required equations.

Arrows are strict in $\mathbb{C}^I$ if, and only if, all their members are strict:

Lemma 5.3 (an *I*-indexed family of $\mathbb{C}$-arrows is strict if, and only if, all its members are strict): Let $\alpha : X \to Y$ be an arrow in the $\mathbf{O}\perp$ category $\mathbb{C}^I$ described above, then

$$\alpha \textit{ is strict} \equiv \forall i \in I, \alpha_i : X_i \to Y_i \textit{ is strict}$$

Proof:

Case ($\Leftarrow$):

Assume α_i is strict for each i in I, then for any family, Z, we know

$$\alpha_i \circ \perp_{Z_i \to X_i} = \perp_{Z_i \to Y_i}$$

Then consider the family

$$\perp_{Z \to X} \overset{def}{=} \left\{ \perp_{Z_i \to X_i} \right\}_{i \in I}$$

which is the least element of $hom_{\mathbb{C}^I}(Z, X)$. We calculate

$\alpha \circ \perp_{Z \to X} = \perp_{Z \to Y}$

$\equiv$ { memberwise equality }

$\forall i \in I, (\alpha \circ \perp_{Z \to X})_i = (\perp_{Z \to Y})_i$

$\equiv$ { composition of families }

$\forall i \in I, \alpha_i \circ (\perp_{Z \to X})_i = (\perp_{Z \to Y})_i$

$\equiv$ { definition of $\perp_{Z \to X}$ }

$\forall i \in I, \alpha_i \circ \perp_{Z_i \to X_i} = (\perp_{Z \to Y})_i$

$\equiv$ { because each α_i is strict }

True

which mean that α is strict.

Case ($\Rightarrow$):

Suppose α is strict, then we must show for each i in I and object A in $\mathbb{C}$:

$$\alpha_i \circ \perp_{A \to X_i} = \perp_{A \to Y_i}$$

Let A and i be given. Then construct the family Z to be equal to X, except in the i-th member, which we set to equal A:

$$\forall j \in I, Z_j \overset{def}{=} \begin{cases} A & \textit{if } j = i \\ X_j & \textit{otherwise.} \end{cases}$$

We can now calculate:

$$\begin{array}{ll}
 & \alpha_i \circ \perp_{A \to X_i} = \perp_{A \to Y_i} \\
\equiv & \quad \{ \text{definition of } Z \} \\
 & \alpha_i \circ \perp_{Z_i \to X_i} = \perp_{Z_i \to Y_i} \\
\Leftarrow & \\
 & \forall j \in I, \alpha_j \circ \perp_{Z_j \to X_j} = \perp_{Z_j \to Y_j} \\
\equiv & \quad \{ \text{composition of families} \} \\
 & \forall j \in I, (\alpha \circ \perp_{Z \to X})_j = (\perp_{Z \to Y})_j \\
\equiv & \quad \{ \text{equality of families} \} \\
 & \alpha \circ \perp_{Z \to X} = \perp_{Z \to Y} \\
\equiv & \quad \{ \text{definition of strict} \} \\
 & \alpha \text{ is strict} \\
\equiv & \quad \{ \text{assumption} \} \\
 & \text{True}
\end{array}$$

So α_i is strict for each i in I.

From which we straightforwardly conclude that

Corollary 5.2:

$(\mathbb{C}^I)_\perp \cong (\mathbb{C}_\perp)^I$

because strict arrows in $\mathbb{C}^I$ are exactly those with all strict members.

We can now state a version of theorems 2.4 and 2.5 for our category $\mathbb{C}^I$ of indexed families:

Theorem 5.2: Let I be a set considered as a *discrete category* and $\mathbb{C}$ a localized **O**$\perp$-category with a terminal object. Let $\mathbb{C}$ be either ω-cocomplete or ω^{op}-complete, then any locally ω-continuous endofunctor $F : \left(\mathbb{C}^I\right)_\perp \to \left(\mathbb{C}^I\right)_\perp$ has an initial algebra. Furthermore, if $\alpha : F.X \to X$ is any F-algebra in $\left(\mathbb{C}^I\right)_\perp$, then the following are equivalent:

(i) α is an initial F-algebra;

(ii) α is an isomorphism and the least fixed-point of

$$f \mapsto \alpha \circ F.f \circ \alpha^{-1}$$

is the identity on X.

Proof: $\mathbb{C}$ is a localized **O**$\perp$-category, and by lemmas 5.2 and A.5, so is $\mathbb{C}^I$. If $\mathbb{C}$ is ω-cocomplete or ω^{op}-complete then so is $\mathbb{C}^I$, and lemma 2.7 tells us that $\left(\mathbb{C}^I\right)_E$ is therefore ω-cocomplete. If $\mathbb{C}$ has a terminal object $\mathbb{1}$ then $\underline{\mathbb{1}}$ is terminal in $\mathbb{C}^I$, and theorem 2.3 tells us that $\underline{\mathbb{1}}$ is initial in $\left(\mathbb{C}^I\right)_E$. Thus $\mathbb{C}^I$ satisfies the conditions of theorem 2.4 and we can apply it to find the initial algebra of F.

Similarly, $\mathbb{C}^I$ satisfies the conditions of theorem 2.5 and therefore statements (i) and (ii) are equivalent.

Of course, we still need to know that the base functors for our algebra families are locally ω-continuous, and for this we can appeal to the results listed in lemma 3.22. In particular, notice that "next" functors are locally ω-continuous because by their definition they are precomposition, for example,

$$nextP \overset{def}{=} (\circ(P{+\!\!+}))$$

and these are already covered by theorem 3.6.

5.8. Initial algebra families in Haskell

We should now justify that the algebra family folds we define in Haskell do correspond to catamorphisms in a functor category $\mathbb{C}^I$. Again we show this for the type of nests, but other data-types follow a similar argument.

We know that the functor for nests

$$F \overset{def}{=} \underline{\{\,\mathbb{1}\,\}} \ddot{+} \underline{\{\,\Omega.A\,\}} \ddot{\times}\ nextP$$

is locally ω-continuous and so we can appeal to theorem 5.2. Then we can form an algebra for this functor from the polymorphic Haskell constructors for nests:

$$\alpha \overset{def}{=} \left\{\mathsf{NilN}_{\Omega.A}\right\} \dot{\triangledown} \left\{\mathsf{ConsN}_{\Omega.A}\right\}$$

The carrier for this algebra is the family $\{\,Nest.\Omega.A\,\}$. We can define this algebra family in Haskell as

```
alpha :: NestAlgFam Nest P a a
alpha = MkNestAlgFam {
     nilnreplace = NilN ,
     consnreplace = ConsN ,
    nextP = alpha
  }
```

Then α is an isomorphism because every member is an isomorphism. To show that α is an initial algebra we thus need only to show that the least fixed-point of

$$f \mapsto \alpha \circ F.f \circ \alpha^{-1}$$

is the identity on $\{\, Nest.\Omega.A \,\}$. Because least fixed-points are computed pointwise, each member of the least fixed-point corresponds to an instance of

```
(foldNestAlgFam alpha) :: Nest a → Nest a
```

for the types a, (P a), (P (P a)), and so on. So if we can show that (foldNestAlgFam alpha) is the identity for the types (Nest a), (Nest (P a)), and so on, then we are done. For finite nests this is easily achieved by induction on the length of the nest. Infinite nests are the least upper bounds of ascending ω-chains of finite approximations. Because (foldNestAlgFam alpha) is the identity on all the finite approximations, it is also the identity on the limit.

5.9. Chapter summary

To give algebra family folds a categorical semantics, algebra families are treated as indexed sets of replacement functions. This allows us to work in a functor category $\mathbb{C}^I$, where I is the indexing set considered as a discrete category. Algebra family folds turn out to be catamorphisms from initial algebras in this category.

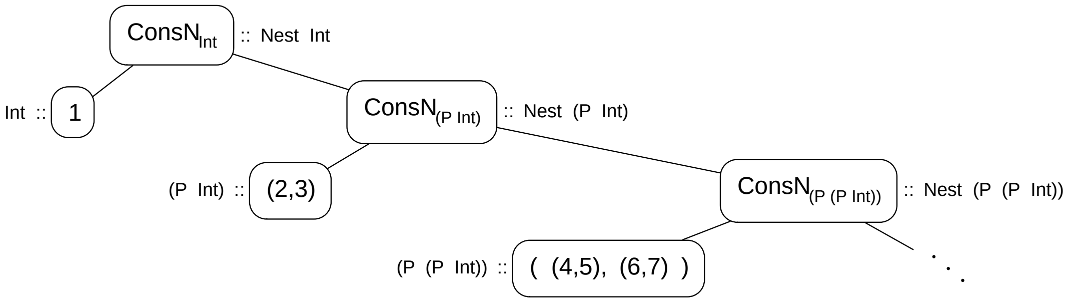
From initiality we can derive transformation laws for algebra family folds, in particular, catamorphism fusion. However, fusion is more difficult to apply than for uniform types because we are now dealing with infinite families of arrows rather than single arrows. Consequently, more work must be done to use the transformation laws, although this can be simplified for special cases.

To prove the existence of initial algebras in $\mathbb{C}^I$, it is possible to re-use the results of chapter 3. In fact, it is possible to simplify some of the results because I is a *discrete* category. The key to the existence of initial algebra families for non-uniform types is establishing the ω-cocontinuity or local ω-continuity of "next" functors, and this is readily achieved.

Chapter 6. Mapping over non-uniform data-types

Mapping is a simpler pattern of recursion than folding, and this is demonstrated by the fact that maps can be expressed as folds. Programmers who know how to derive maps for uniform types should have little difficulty in coming up with the appropriate map operators for non-uniform types also. We use nests, as usual, for our motivating example.

The problem presented by non-uniform data-types is that the structure of a value changes with recursion depth, and to be able to successfully map a function across such a structure, the function being mapped must change accordingly. Consider the first few elements of a nest structure:


ConsN$_{Int}$ :: Nest Int
Int :: 1
ConsN$_{(P\ Int)}$ :: Nest (P Int)
(P Int) :: (2,3)
ConsN$_{(P\ (P\ Int))}$ :: Nest (P (P Int))
(P (P Int)) :: ((4,5), (6,7))

and suppose we want to map the function

$$\text{odd} :: \text{Int} \rightarrow \text{Bool}$$

that returns True only if its argument is an odd integer, across the nest. Processing begins at the top of the diagram. The function odd can be successfully applied to the left-hand child because it has integer type. However, when we wish to process the rest of the nest by continuing down the right-hand branch, we see that we are no longer dealing with a nest of Int's, but instead a nest of (P Int)'s. We cannot simply use our function odd as the argument to a recursive call to map because it would not have the correct type. We need a way of changing our argument odd :: Int → Bool to operate on (P Int) values, and then (P (P Int)) values, and so on. Fortunately, this is easy provided that P is a functor, that is, that it has a map function itself:

```
mapP :: (a → b) → P a → P b
```

Then we can define mapNest:

```
mapNest f NilN = NilN
mapNest f (ConsN x xs) = ConsN (f x) (mapNest (mapP f) xs)
```

The mapP function takes care of changing the mapNest argument type. Notice that we are again using polymorphic recursion, and must therefore explicitly give the type signature:

```
mapNest :: (a → b) → (Nest a → Nest b)
```

Given that mapping is simpler than folding, and that it is relatively easy to produce a useful map operator for a non-uniform type, why do we treat the more difficult case of folding first? There are two reasons:

- we would like to give a systematic derivation of map operators for arbitrary linear non-uniform data-types, and this can be done by expressing maps as algebra family folds;
- we would also like to prove the useful laws about maps such as their preservation of composition and identities and the way they interact with folds; if maps are expressed as algebra family folds then we can use the general proof principles relating to algebra family catamorphisms to try to prove these laws.

Thus we will re-use the framework we have developed for folding over non-uniform types to capture and reason about maps. This is in keeping with the traditional treatment of maps over uniform types in Squiggol.

6.1. Mapping using algebra family folds

Recall from section 2.3 that in order to be able to express maps over uniform types as folds we first had to introduce some way of parameterizing our data-types. This was done by abstracting out the parameter type and then taking the initial algebras of sectioned bifunctors. Then maps could be expressed as catamorphisms between these initial algebras. We take exactly the same approach for our algebra family folds.

6.1.1. Parameterizing initial algebra families

Although our initial algebra families are polymorphic in some sense, in that they contain multiple instances of the constructors, they are geared towards constructing values of a *particular instance* of a non-uniform type. For instance, the algebra family

$$\begin{tikzcd}
\{\mathbb{1}\} \dot{+} \{\Omega.Int\} \dot{\times} \{Nest.P.\Omega.Int\} \arrow[d, "\{\mathsf{NilN}_{\Omega.Int}\} \dot{\triangledown} \{\mathsf{ConsN}_{\Omega.Int}\}"] \\
\{Nest.\Omega.Int\}
\end{tikzcd}$$

contains all the constructor instances you could ever need to build a nest of integers, but is of absolutely no help in building a nest of booleans.

Recall the base functor for the initial algebra family for nests of some type A:

$$\begin{tikzcd}
F : \mathbb{C}^I \arrow[rr] & & \mathbb{C}^I \\
X \arrow[dd, "\alpha"'] & & \{\mathbb{1}\} \dot{+} \{\Omega.A\} \dot{\times} nextP.X \arrow[dd, "Id \dot{+} Id \dot{\times} nextP.\alpha"] \\
& \mapsto & \\
Y & & \{\mathbb{1}\} \dot{+} \{\Omega.A\} \dot{\times} nextP.Y
\end{tikzcd}$$

where the implicit indexing set I is $\{P\}^*$. In exactly the same way as we did for uniform types, we can abstract out occurrences of the fixed type A to give us a bifunctor:

$$\begin{tikzcd}
\ddagger : \mathbb{C} \times \mathbb{C}^I \arrow[r] & \mathbb{C}^I \\
(A, X) \arrow[d, "{(f, \alpha)}"'] & \{\, \mathbb{1} \,\} \dot{+} \{\, \Omega.A \,\} \dot{\times} nextP.X \arrow[d, "Id \dot{+} \{\, \Omega.f \,\} \dot{\times} nextP.\alpha"] \\
(B, Y) \arrow[r, phantom, "\mapsto"] & \{\, \mathbb{1} \,\} \dot{+} \{\, \Omega.B \,\} \dot{\times} nextP.Y
\end{tikzcd}$$

Then we see that if we section $\ddagger$ with any type A it becomes the base functor for nests of that type.

6.1.2. Constructing the algebra family fold for mapping

Suppose we have an arrow $f : A \to B$ in $\mathbb{C}$, and we know that the functors $(A\ddagger)$ and $(B\ddagger) : \mathbb{C}^I \to \mathbb{C}^I$ have initial algebras

$$in^{A\ddagger} : A \ddagger \mu(A\ddagger) \to \mu(A\ddagger)$$

and

$$in^{B\ddagger} : B \ddagger \mu(B\ddagger) \to \mu(B\ddagger)$$

respectively. Then we define the mapping of f across a $\mu(A\ddagger)$ value by the algebra family fold:

$$\begin{tikzcd}
A \ddagger \mu(A\ddagger) \arrow[r, "Id \ddagger (\!| in^{B\ddagger} \circ (f \ddagger Id) |\!)^{A\ddagger}"] \arrow[dd, "in^{A\ddagger}"'] & A \ddagger \mu(B\ddagger) \arrow[d, "f \ddagger Id"] \\
 & B \ddagger \mu(B\ddagger) \arrow[d, "in^{B\ddagger}"] \\
\mu(A\ddagger) \arrow[r, dotted, "(\!| in^{B\ddagger} \circ (f \ddagger Id) |\!)^{A\ddagger}"'] & \mu(B\ddagger)
\end{tikzcd} \tag{6.1.1}$$

Let us write this catamorphism as $\textcircled{$\ddagger$}.f$:

$$\textcircled{$\ddagger$}.f \overset{def}{=} (\!|\, in^{B\ddagger} \circ (f \ddagger Id) \,|\!)^{A\ddagger}$$

This is the same construction as we used in section 2.3.1 to express maps over uniform types as folds; the difference here is that we are working with *families* of arrows and not just single arrows. Of course, the resulting catamorphism is an I-indexed *family* of arrows, and we only want the first member of that family:

$$(\textcircled{$\ddagger$}.f)_{Id} : \mu(A\ddagger)_{Id} \to \mu(B\ddagger)_{Id}$$

Let us expand diagram 6.1.1 for the case when $\ddagger$ is the bifunctor for nests:

$$\begin{tikzcd}
\{\mathbb{1}\} \dotplus \{\Omega.A\} \dot{\times} nextP.\{Nest.\Omega.A\} \arrow[r, "Id \dotplus Id \dot{\times} nextP.(\textcircled{$\ddagger$}.f)"] \arrow[dd, "\{\mathsf{NilN}_{\Omega.A}\} \dot{\triangledown} \{\mathsf{ConsN}_{\Omega.A}\}"'] & \{\mathbb{1}\} \dotplus \{\Omega.A\} \dot{\times} nextP.\{Nest.\Omega.B\} \arrow[d, "Id \dotplus \{\Omega.f\} \dot{\times} nextP.Id"] \\
 & \{\mathbb{1}\} \dotplus \{\Omega.B\} \dot{\times} nextP.\{Nest.\Omega.B\} \arrow[d, "\{\mathsf{NilN}_{\Omega.B}\} \dot{\triangledown} \{\mathsf{ConsN}_{\Omega.B}\}"] \\
\{Nest.\Omega.A\} \arrow[r, dotted, "\textcircled{$\ddagger$}.f"'] & \{Nest.\Omega.B\}
\end{tikzcd}$$

We can split this diagram into the two equations:

$$\textcircled{$\ddagger$}.f \circ \left\{\mathsf{NilN}_{\Omega.A}\right\} = \left\{\mathsf{NilN}_{\Omega.B}\right\} \qquad (6.1.2)$$

and

$$\textcircled{$\ddagger$}.f \circ \left\{\mathsf{ConsN}_{\Omega.A}\right\} = \left\{\mathsf{ConsN}_{\Omega.B}\right\} \circ \left(\left\{\Omega.f\right\} \dot{\times} nextP.(\textcircled{$\ddagger$}.f)\right) \qquad (6.1.3)$$

Then we can observe the behaviour of $(\textcircled{$\ddagger$}.f)_{Id}$ when applied to the constructors NilN_A and ConsN_A respectively:

$(\textcircled{$\ddagger$}.f)_{Id} \circ \mathsf{NilN}_A$

$=$ { member selection }

$$
\begin{aligned}
& (\textcircled{\ddagger}.f)_{Id} \circ \left\{ \mathsf{NilN}_{\Omega.A} \right\}_{Id} \\
= & \qquad \{ \text{memberwise composition of families} \} \\
& \left(\textcircled{\ddagger}.f \circ \left\{ \mathsf{NilN}_{\Omega.A} \right\} \right)_{Id} \\
= & \qquad \{ \text{equation 6.1.2} \} \\
& \left\{ \mathsf{NilN}_{\Omega.B} \right\}_{Id} \\
= & \qquad \{ \text{member selection} \} \\
& \mathsf{NilN}_B
\end{aligned}
$$

As usual, the ConsN case is more involved:

$$
\begin{aligned}
& (\textcircled{\ddagger}.f)_{Id} \circ \mathsf{ConsN}_A \\
= & \qquad \{ \text{member selection} \} \\
& (\textcircled{\ddagger}.f)_{Id} \circ \left\{ \mathsf{ConsN}_{\Omega.A} \right\}_{Id} \\
= & \qquad \{ \text{memberwise composition of families} \} \\
& \left(\textcircled{\ddagger}.f \circ \left\{ \mathsf{ConsN}_{\Omega.A} \right\} \right)_{Id} \\
= & \qquad \{ \text{equation 6.1.3} \} \\
& \left(\left\{ \mathsf{ConsN}_{\Omega.B} \right\} \circ \left(\left\{ \Omega.f \right\} \dot{\times}\, nextP.(\textcircled{\ddagger}.f) \right) \right)_{Id} \\
= & \qquad \{ \text{memberwise composition of families} \} \\
& \left\{ \mathsf{ConsN}_{\Omega.B} \right\}_{Id} \circ \left(\left\{ \Omega.f \right\} \dot{\times}\, nextP.(\textcircled{\ddagger}.f) \right)_{Id} \\
= & \qquad \{ \text{lifted product} \} \\
& \left\{ \mathsf{ConsN}_{\Omega.B} \right\}_{Id} \circ \left(\left\{ \Omega.f \right\}_{Id} \times (nextP.(\textcircled{\ddagger}.f))_{Id} \right) \\
= & \qquad \{ \text{member selection} \} \\
& \mathsf{ConsN}_B \circ \left(f \times (nextP.(\textcircled{\ddagger}.f))_{Id} \right) \\
= & \qquad \{ \text{definition of } nextP \} \\
& \mathsf{ConsN}_B \circ \left(f \times (\textcircled{\ddagger}.f)_P \right)
\end{aligned}
$$

So $(\textcircled{\ddagger}.f)_{Id}$ replaces NilN_A and ConsN_A by NilN_B and ConsN_B respectively, while applying f

to the first argument and recursively processing the second argument to ConsN_A. The recursive call to the map is $(\textcircled{\ddagger}.f)_P$, and we can calculate the result of applying $(\textcircled{\ddagger}.f)_P$ to the next instances of the constructors we might meet — $\mathsf{NilN}_{P.A}$ and $\mathsf{ConsN}_{P.A}$:

$$
\begin{aligned}
& (\textcircled{\ddagger}.f)_P \circ \mathsf{NilN}_{P.A} \\
= & \\
& \mathsf{NilN}_{P.B}
\end{aligned}
$$

and

$$
\begin{aligned}
& (\textcircled{\ddagger}.f)_P \circ \mathsf{ConsN}_{P.A} \\
= & \\
& \mathsf{ConsN}_{P.B} \circ \Big(P.f \times (\textcircled{\ddagger}.f))_{PP} \Big)
\end{aligned}
$$

The constructors are again replaced with different instances, and the second argument to $\mathsf{ConsN}_{P.A}$ is recursively processed, but this time $P.f$ is applied to the first argument, instead of just f. In subsequent recursive calls, the function applied will be $P.P.f$ then $P.P.P.f$, and so on. This is clearly the same behaviour as the map function we naively defined:

```
mapNest f NilN = NilN
mapNest f (ConsN x xs) =
        ConsN (f x) (mapNest (mapP f) xs)
```

which replaces constructor instances and successively applies f, (mapP f), (mapP (mapP f)), and so on in the recursive calls.

Thus for any non-uniform data-type, whose initial algebra families are initial algebras of sections of a bifunctor $\ddagger : \mathbb{C} \times \mathbb{C}^I \to \mathbb{C}^I$, we can map a function $f : A \to B$ by using the catamorphism:

$$\textcircled{\ddagger}.f \overset{def}{=} \llparenthesis\, in^{B\ddagger} \circ (f \ddagger Id) \,\rrparenthesis^{A\ddagger}$$

and this is analogous to the way in which maps are defined as catamorphisms over *uniform* types.

6.2. Proving properties of maps

Maps usually satisfy certain standard properties such as preservation of identities and composition. Because we have expressed our maps over non-uniform types as algebra family folds, using the same construction as we would for uniform types, we can prove the properties in exactly the same way. First we prove that maps preserve identities:

> **Lemma 6.1:** Let $\ddagger : \mathbb{C} \times \mathbb{C}^I \to \mathbb{C}^I$ be a bifunctor such that $(A\,\ddagger) : \mathbb{C}^I \to \mathbb{C}^I$ has an initial algebra
>
> $$in^{A\ddagger} : A \ddagger \mu(A\,\ddagger) \to \mu(A\,\ddagger)$$
>
> then
>
> $$\textcircled{$\ddagger$}.Id_A = Id_{\mu(A\ddagger)}$$

Proof: We calculate

$$
\begin{array}{ll}
 & \textcircled{$\ddagger$}.Id_A \\
= & \quad \{\text{ definition of } \textcircled{$\ddagger$} \ \} \\
 & \llparenthesis\, in^{A\ddagger} \circ (Id_A \ddagger Id) \,\rrparenthesis^{A\ddagger} \\
= & \quad \{\ \ddagger \text{ is a bifunctor } \} \\
 & \llparenthesis\, in^{A\ddagger} \circ Id \,\rrparenthesis^{A\ddagger} \\
= & \quad \{\text{ identity } \} \\
 & \llparenthesis\, in^{A\ddagger} \,\rrparenthesis^{A\ddagger} \\
= & \quad \{\text{ catamorphism self (2.4.2) } \} \\
 & Id_{\mu(A\ddagger)}
\end{array}
$$

To prove that maps preserve composition we use catamorphism fusion:

Lemma 6.2: Let $\ddagger : \mathbb{C} \times \mathbb{C}^I \to \mathbb{C}^I$ be a bifunctor such that $(A\ddagger), (B\ddagger), (C\ddagger) : \mathbb{C}^I \to \mathbb{C}^I$ all have initial algebras, and let $f : A \to B$ and $g : B \to C$ be arrows in $\mathbb{C}$. Then

$$\textcircled{\ddagger}.(g \circ f) = \textcircled{\ddagger}.g \circ \textcircled{\ddagger}.f$$

Proof: Central to this proof is the commutativity of the following fusion diagram:

$$\begin{tikzcd}
A \ddagger \mu(A\ddagger) \arrow[r, "Id \ddagger \textcircled{\ddagger}.f"] \arrow[ddd, "in^{A\ddagger}"'] & A \ddagger \mu(B\ddagger) \arrow[r, "Id \ddagger \textcircled{\ddagger}.g"] \arrow[d, "f \ddagger Id"] & A \ddagger \mu(C\ddagger) \arrow[d, "f \ddagger Id"] \\
 & B \ddagger \mu(B\ddagger) \arrow[r, "Id \ddagger \textcircled{\ddagger}.g"] \arrow[dd, "in^{B\ddagger}"] & B \ddagger \mu(C\ddagger) \arrow[d, "g \ddagger Id"] \\
 & & C \ddagger \mu(C\ddagger) \arrow[d, "in^{C\ddagger}"] \\
\mu(A\ddagger) \arrow[r, "\textcircled{\ddagger}.f"'] & \mu(B\ddagger) \arrow[r, "\textcircled{\ddagger}.g"'] & \mu(C\ddagger)
\end{tikzcd}$$

The top-right-hand square can be easily shown to commute by using the fact that $\ddagger$ is a bifunctor. The other two squares commute because they are the definitions of $\textcircled{\ddagger}.f$ and $\textcircled{\ddagger}.g$. Then we can reason

$$\begin{aligned}
& \textcircled{\ddagger}.g \circ \textcircled{\ddagger}.f \\
= \quad & \{ \text{ definition of } \textcircled{\ddagger}.f \ \} \\
& \textcircled{\ddagger}.g \circ (\!| in^{A\ddagger} \circ (f \ddagger Id) |\!)^{A\ddagger} \\
= \quad & \{ \text{ catamorphism fusion } \}
\end{aligned}$$

$$
\begin{aligned}
& \llparenthesis\, in^{C\ddagger} \circ (f \ddagger Id) \circ (g \ddagger Id) \,\rrparenthesis^{A\ddagger} \\
= \quad & \{ \ddagger \text{ is a bifunctor} \} \\
& \llparenthesis\, in^{C\ddagger} \circ ((f \circ g) \ddagger Id) \,\rrparenthesis^{A\ddagger} \\
= \quad & \{ \text{definition of } \textcircled{$\ddagger$}.(g \circ f) \} \\
& \textcircled{$\ddagger$}.(g \circ f)
\end{aligned}
$$

and we are done.

These results are proved in exactly the same way that they would be proved for maps over uniform types — the only difference is that we are working in the functor category $\mathbb{C}^I$.

Another standard Squiggol rule is that a map followed by a catamorphism can be combined into a single catamorphism:

Lemma 6.3: Let $\ddagger : \mathbb{C} \times \mathbb{C}^I \to \mathbb{C}^I$ be a bifunctor such that $(A\ddagger), (B\ddagger) : \mathbb{C}^I \to \mathbb{C}^I$ have initial algebras. Let $f : A \to B$ be an arrow in $\mathbb{C}$ and $\alpha : B \ddagger C \to C$ a $(B\ddagger)$-algebra in $\mathbb{C}^I$. Then

$$\llparenthesis\, \alpha \,\rrparenthesis^{B\ddagger} \circ \textcircled{$\ddagger$}.f = \llparenthesis\, \alpha \circ (f \ddagger Id) \,\rrparenthesis^{A\ddagger}$$

Proof: Similar to lemma 6.2. The fusion diagram in this case is:

$$\begin{tikzcd}
A \ddagger \mu(A\ddagger) \arrow[r, "Id \ddagger \textcircled{$\ddagger$}.f"] \arrow[dd, "in^{A\ddagger}"'] & A \ddagger \mu(B\ddagger) \arrow[r, "Id \ddagger \llparenthesis \alpha \rrparenthesis^{B\ddagger}"] \arrow[d, "f \ddagger Id"] & A \ddagger C \arrow[d, "f \ddagger Id"] \\
 & B \ddagger \mu(B\ddagger) \arrow[r, "Id \ddagger \llparenthesis \alpha \rrparenthesis^{B\ddagger}"] \arrow[d, "in^{B\ddagger}"] & B \ddagger C \arrow[d, "\alpha"] \\
\mu(A\ddagger) \arrow[r, "\textcircled{$\ddagger$}.f"'] & \mu(B\ddagger) \arrow[r, "\llparenthesis \alpha \rrparenthesis^{B\ddagger}"'] & C
\end{tikzcd}$$

6.3. Chapter summary

In contrast with folds, it is not difficult to invent map combinators for non-uniform types, provided that the modifying functors themselves have map operations. However, to establish calculational laws about maps, it is helpful to express them as algebra family folds. To do this, we must first *parameterize* our initial algebra families, then the construction of maps as algebra family folds proceeds in the same way as maps are defined for uniform types. The calculational laws for maps can then be proved in the same way as for uniform types.

Chapter 7. Conclusion

We now summarize the work in this thesis and present some opportunities for further work.

7.1. Summary

In this thesis we have looked at the problem of extending the common structured recursion operators *map* and *fold* to linear non-uniformly recursive data-types.

In chapter 2 we reviewed the standard categorical treatment of uniform recursive data-types as initial algebras and presented two standard tools for proving the existence of initial algebras — the generalized Kleene fixed-point theorem and the work of Wand, Smyth, Plotkin and others in order-enriched categories.

In chapter 3 we considered initial algebras in *functor categories* and saw that they provide a form of fold operator for non-uniform types. However, this fold operator is limited in its applications because it works with *natural transformations* or *parametric polymorphic* functions and can only be used to define operations that are parametric polymorphic. The contribution of chapter 3 is the lifting of the tools explained in chapter 2 to show the existence of initial algebras in functor categories.

In chapter 4 we described an alternative style of folding over non-uniform data-types based on the slogan that “folds replace constructors”. We approached the problem more from a programmer’s point of view and showed how to design fold combinators in Haskell that take infinite *algebra families* as arguments. The algebra families contain the replacement functions for the constructor instances in a value of a non-uniformly recursive type. These *algebra family folds* generalized the functor category folds from chapter 3, and overcome the limitations they faced because of naturality requirements.

Chapter 5 develops a categorical semantics for algebra family folds in terms of indexed families of arrows from a base category. In this setting, algebra family folds are *catamorphisms* from *initial algebra families*, and we used initiality to formulate the corresponding program transformation laws for algebra family folds. We also looked at conditions for the existence of initial algebra families, again by lifting the tools described in chapter 2.

Finally, we used algebra family folds in chapter 6 to define *maps* over non-uniform data-types. By expressing maps as catamorphisms from initial algebra families, we proved the standard properties about preservation of identity and composite functions.

7.2. Discussion

As we have seen, folding over non-uniform data-types by replacing constructors is qualitatively more complex than folding over uniform types because of the infinite number of constructor instances that must be considered. Although we can hide the extra complexity in the recursion by providing the structured recursion operators map and fold, in the case of fold, the programmer is still faced with the problem of constructing an appropriate algebra family. We can alleviate this problem somewhat by providing simpler versions of algebra families and additional combinators for constructing algebra families, but this must currently be done on an ad-hoc basis by analyzing a particular type definition and seeing where functors commute.

Algebra family folds can be used for many applications on non-uniform types, including *all* those that can be captured as higher-order catamorphisms. The reason for this generality is because we allow our fold combinators to take arbitrary algebra families as arguments — we do not make any restrictions on how such algebra families should be constructed. Although this is good in that it makes algebra family folds very expressive, from the program transformation point of view, the calculational laws are not as immediately useful as the corresponding laws for uniform types because they are considerably more difficult to apply. To apply the fusion law to a uniform type, one must simply show that the right-hand square commutes in the fusion diagram:

$$\begin{tikzcd}
F.\mu F \arrow[r, "F.\llparenthesis \alpha \rrparenthesis"] \arrow[d, "in^F"'] & F.A \arrow[r, "F.h"] \arrow[d, "\alpha"] & F.B \arrow[d, "\beta"] \\
\mu F \arrow[r, dotted, "\llparenthesis \alpha \rrparenthesis"'] & A \arrow[r, "h"'] & B
\end{tikzcd}$$

that is, prove a single equality between two arrows. For non-uniform types, however, we are dealing with *infinite families* of arrows, and the above single equation translates into infinitely many equations that must be proved before fusion can be applied. It then becomes necessary to exploit any available extra knowledge about how the algebra families are constructed in order to prove the equations. We saw an example of this when we used fusion to prove the laws for mapping in chapter 6; the algebra families used for mapping were particularly simple. In general though, a solution to this problem would seem to depend on finding a systematic way of constructing algebra families for a large class of non-uniform types and then using knowledge about the construction of the algebra families to simplify the proof obligations for the transformation laws. There would seem to be a trade-off between expressiveness and amenability to transformation. It is hoped that through more experience of using non-uniform data-types, that a comfortable balance can be found.

To some extent we have restricted our investigation to recursion combinators that can be implemented in currently available extensions to Haskell. This has influenced the design of our combinators and, consequently, our categorical semantics for them. It is hoped that as non-uniform data-types mature and become more widely used, programming language type systems will evolve to incorporate non-uniform types more naturally, and it will be easier to provide an elegant formalism for reasoning about them on an equal footing with uniform types. Whether this is best done by trying to produce a general initial algebra semantics or by following a different approach (such as [Hinze99]) remains to be seen.

7.3. Future work

We have described the foundations for using algebra family folds over non-uniform data-types, but there are several opportunities for further work, notably methods of constructing algebra family folds and algebra family folds over higher-order non-uniform data-types. We now briefly discuss these and other issues.

7.3.1. Constructing algebra families

We have mentioned the problems of constructing algebra families and there is need for a better understanding of how algebra families can be produced for particular data-types and how sensible and useful combinators can be given to aid construction. Some recent work on *commuting data-types* [HoogendijkB97] and *functor pullers* [Meertens98] may prove relevant to this problem as it is often necessary to “pull” or commute the modifying functors in order to be able to successfully grow algebra families.

There is also the question of efficiency — different algebra families may give folds that produce the same results, but one may be more efficient than the other. Is it possible to transform the less efficient family into the more efficient one?

7.3.2. Higher-order non-uniform data-types

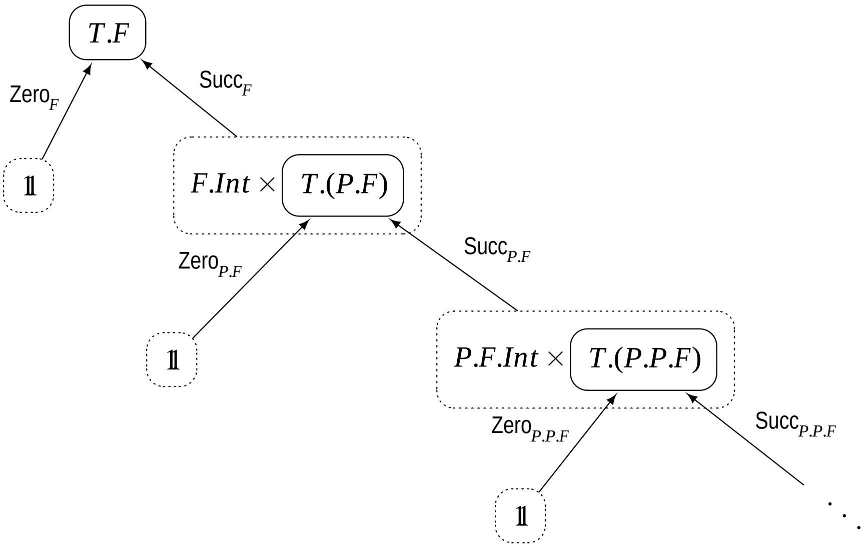
In this thesis we have concentrated on simple non-uniform data-types that are parameterized by *type*. However, researchers [Hinze99, Okasaki99] have recently begun exploring applications of *higher-order* non-uniform data-types — those that are parameterized not by type, but by *type constructor*. A simple example would be:

```
newtype P f a = MkP (f a, f a)
data T f = Zero
         | Succ (f Int) (T (P f))
```

The parameter f in the definition of the type T takes values of kind $(* \to *)$. Also, the modifying functor P now applies to *type constructors*, so both T and P have kind $((* \to *) \to (* \to *))$.

Reasons for choosing to use a higher-order non-uniform type may be to better match a particular data-type design style [Hinze98], or it may be that it is necessary to move to higher-order non-uniform types in order to impose the desired invariants, as Okasaki found when trying to encode square matrices [Okasaki99].

Is it possible to fold over higher-order non-uniform types using algebra family folds? We can certainly carry out the process described in chapter 4 — if we fix the parameter *F* then we can examine the possible constructor instances in exactly the same way:



From this we can derive a type for algebra families:

```
data TAlgFam r s f g = MkTAlgFam {
     zeroreplace :: r g ,
     succreplace :: (f Int) → r (s g) → r g ,
     nextP :: TAlgFam r s (P f) (s g)
  }
```

and the fold combinator is straightforward:

```
foldTAlgFam :: TAlgFam r s f g → T f → r g
foldTAlgFam fam Zero = zeroreplace fam
foldTAlgFam fam (Succ x xs) =
                    (succreplace fam) x (foldTAlgFam (nextP fam) xs)
```

However, algebra families for higher-order non-uniform types may turn out to be more difficult to construct and place more strain on the type systems, perhaps necessitating the introduction of *rank-3* (or higher) polymorphism.

7.3.3. Other recursion combinators

Although we have concentrated on initial algebras and folds or catamorphisms, the dual notions of *terminal coalgebras* and *unfolds* or *anamorphisms* are finding increasingly many applications in computer science [GibbonsJ98, Hutton98, MeijerFP91]. Unfolds on non-uniform data-types can be defined using the same ideas that we have used to define functor category or algebra family folds. In fact, Fokkinga and Meijer's main theorem (theorem 2.4) which we use in our order-enriched functor categories (theorems 3.5 and 5.2), establishes not only the existence of *initial algebras*, but also *terminal coalgebras*, and shows that they coincide. Then it is possible to define a new recursion operator called a *hylomorphism* [MeijerFP91] that generalizes both catamorphisms and anamorphisms, and can be used in deforestation techniques [TakanoM95].

7.3.4. Fibrational semantics

The categorical semantics we developed in chapter 5 was fairly concrete in that it modelled the algebra families as families indexed by a fixed set I. The "next" functors that move down the tree structure of the algebra families could be modelled as re-indexings inside the same fixed indexing set, and this allowed us to perform all our constructions conveniently inside a functor category $\mathbb{C}^I$, re-using our previous results about functor categories.

Fibrations [Bénabou85, Phoa92, Jacobs99] or *indexed categories* [Taylor86, TarleckiBG91] offer a more general framework for dealing with indexed families. The fact that our constructions were carried out in a functor category $\mathbb{C}^I$ means, in the language of fibrations, that we were working within a single *fibre*, denoted $\mathit{Fam}(\mathbb{C})_I$, of the fibration $\mathit{Fam}(\mathbb{C})$ of set-indexed families of $\mathbb{C}$-objects. However, the "next" functors can more accurately be described as re-indexings that move *between* different fibres, and this approach may give a more satisfactory semantics. Indeed, in the fibrational setting it may be possible to give semantics to more bizarre recursive types in which the indexing sets are not simple sets of the form $\left\{ M_1, \ldots, M_n \right\}^*$.

The main drawback with the fibrational approach is that the theory of fibrations is fairly technical and is not yet widely known in the computer science community, even though fibrations seem to have many applications there, particularly in polymorphic type theory [Taylor86, Jacobs99].

Appendix A. Long proofs

A.1. Regular types in functor categories

Regular parameterized types form initial algebras in the endofunctor category:

Theorem A.1:

Let $\mathbb{C}$ be a category and $\ddagger : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ a bifunctor. If, for every object A in $\mathbb{C}$, the sectioned functor $(A\ddagger) : \mathbb{C} \to \mathbb{C}$ has an initial algebra,

$$in^{A\ddagger} : A \ddagger \mu(A\ddagger) \to \mu(A\ddagger)$$

Then these initial algebras form the components of an initial algebra of the functor $(Id_{\mathbb{C}} \dot{\ddagger}) : \mathbb{C}^{\mathbb{C}} \to \mathbb{C}^{\mathbb{C}}$ with the type functor $\textcircled{\ddagger}$ as the carrier:

$$in^{Id_{\mathbb{C}}\dot{\ddagger}} : Id_{\mathbb{C}} \dot{\ddagger} \textcircled{\ddagger} \underset{\bullet}{\to} \textcircled{\ddagger}$$

$$(in^{Id_{\mathbb{C}}\dot{\ddagger}})_A \overset{def}{=} in^{A\ddagger}$$

Proof: Suppose $in^{A\ddagger} : A \ddagger \mu(A\ddagger) \to \mu(A\ddagger)$ exists for each A in $\mathbb{C}$. Then we can use them to construct a natural transformation $\alpha : Id_{\mathbb{C}} \dot{\ddagger} \textcircled{\ddagger} \underset{\bullet}{\to} \textcircled{\ddagger}$. To see that the types are correct, for any object A in $\mathbb{C}$, the component of α must have type:

$$\begin{array}{ll} & \alpha_A : (Id_{\mathbb{C}} \dot{\ddagger} \textcircled{\ddagger}).A \to \textcircled{\ddagger}.A \\ \equiv & \quad \{ \text{lifted } \ddagger \} \end{array}$$

$$
\begin{aligned}
& \alpha_A : Id_{\mathbb{C}}.A \ddagger \textcircled{$\ddagger$}.A \to \textcircled{$\ddagger$}.A \\
\equiv \quad & \qquad \{\text{ identity, definition of } \textcircled{$\ddagger$} \} \\
& \alpha_A : A \ddagger \mu(A\ddagger) \to \mu(A\ddagger)
\end{aligned}
$$

So we define $\alpha_A \overset{def}{=} in^{A\ddagger}$. We must also show that α is natural, that is, given any $f : A \to B$ in $\mathbb{C}$, that

$$\alpha_B \circ (f \ddagger \textcircled{$\ddagger$}.f) = \textcircled{$\ddagger$}.f \circ \alpha_A$$

This we calculate:

$$
\begin{aligned}
& \textcircled{$\ddagger$}.f \circ \alpha_A \\
= \quad & \qquad \{\text{ definitions of } \textcircled{$\ddagger$} \text{ and } \alpha \} \\
& \llparenthesis in^{B\ddagger} \circ f \ddagger Id \rrparenthesis_{A\ddagger} \circ in^{A\ddagger} \\
= \quad & \qquad \{\text{ catamorphism }\} \\
& in^{B\ddagger} \circ (f \ddagger Id) \circ \left(Id_A \ddagger \llparenthesis in^{B\ddagger} \circ f \ddagger Id \rrparenthesis_{A\ddagger} \right) \\
= \quad & \qquad \{\text{ definition of } \textcircled{$\ddagger$} \} \\
& in^{B\ddagger} \circ (f \ddagger Id) \circ (Id_A \ddagger \textcircled{$\ddagger$}.f) \\
= \quad & \qquad \{\text{ definition of } \alpha, \ddagger \text{ a functor }\} \\
& \alpha_B \circ (f \ddagger \textcircled{$\ddagger$}.f)
\end{aligned}
$$

So α is natural.

The natural transformation α is clearly an $(Id_{\mathbb{C}}\dot{\ddagger})$-algebra, but we must show that it is initial. Let $\beta : Id_{\mathbb{C}} \dot{\ddagger} K \underset{\cdot}{\longrightarrow} K$ be any other $(Id_{\mathbb{C}}\dot{\ddagger})$-algebra, then we must show that there is a unique mediating morphism $h : \textcircled{$\ddagger$} \underset{\cdot}{\longrightarrow} K$ such that

$$\begin{tikzcd}
Id_{\mathbb{C}} \dot{\ddagger} \textcircled{\ddagger} \arrow[r, "Id \dot{\ddagger} h"] \arrow[d, "\alpha"] & Id_{\mathbb{C}} \dot{\ddagger} K \arrow[d, "\beta"] \\
\textcircled{\ddagger} \arrow[r, "h"] & K
\end{tikzcd}$$

First we show the existence of h. For each A in $\mathbb{C}$ we need a component $h_A : \textcircled{\ddagger}.A \to K.A$. Simply define $h_A \overset{def}{=} \llparenthesis \beta_A \rrparenthesis_{A\ddagger}$. We must show that h so defined is natural, that is, for any $f : A \to B$ in $\mathbb{C}$,

$$h_B \circ \textcircled{\ddagger}.f = K.f \circ h_A$$

We use catamorphism fusion twice in the calculation. The first time in:

$$
\begin{array}{ll}
 & K.f \circ h_A \\
= & \quad \{ \text{definition of } h \} \\
 & K.f \circ \llparenthesis \beta_A \rrparenthesis_{A\ddagger} \\
= & \quad \{ \text{catamorphism fusion — see diagram below} \} \\
 & \llparenthesis \beta_B \circ (f \ddagger Id) \rrparenthesis_{A\ddagger}
\end{array}
$$

The fusion diagram is

$$\begin{tikzcd}
A \ddagger \textcircled{\ddagger}.A \arrow[r, "Id \ddagger \llparenthesis \beta_A \rrparenthesis_{A\ddagger}"] \arrow[dd, "\alpha_A = in^{A\ddagger}"] & A \ddagger K.A \arrow[r, "Id \ddagger K.f"] \arrow[dd, "\beta_A"] & A \ddagger K.B \arrow[d, "f \ddagger Id"] \\
 & & B \ddagger K.B \arrow[d, "\beta_B"] \\
\textcircled{\ddagger}.A \arrow[r, "\llparenthesis \beta_A \rrparenthesis_{A\ddagger}"] & K.A \arrow[r, "K.f"] & K.B
\end{tikzcd}$$

and the right-hand square commutes by the naturality of β:

$$\begin{array}{ll}
 & K.f \circ \beta_A \\
= & \{\ \text{naturality of } \beta\ \} \\
 & \beta_B \circ (Id_{\mathbb{C}} \mathbin{\dot{\ddagger}} K).f \\
= & \{\ \text{lifted bifunctor}\ \} \\
 & \beta_B \circ (f \ddagger K.f) \\
= & \{\ \ddagger \text{ a bifunctor}\ \} \\
 & \beta_B \circ (f \ddagger K.f) \circ (Id \ddagger K.f)
\end{array}$$

We use catamorphism fusion a second time in:

$$\begin{array}{ll}
 & h_\beta \circ \textcircled{$\ddagger$}.f \\
= & \{\ \text{definitions of } h \text{ and } \textcircled{$\ddagger$}\ \} \\
 & \llparenthesis \beta_B \rrparenthesis_{B\ddagger} \circ \llparenthesis in^{B\ddagger} \circ (f \ddagger Id) \rrparenthesis_{A\ddagger} \\
= & \{\ \text{catamorphism fusion — see diagram below}\ \} \\
 & \llparenthesis \beta_B \circ (f \ddagger Id) \rrparenthesis_{A\ddagger}
\end{array}$$

The fusion diagram being:

$$\begin{tikzcd}
A \ddagger \textcircled{$\ddagger$}.A \arrow[r, "Id \ddagger \textcircled{$\ddagger$}.f"] \arrow[dd, "\alpha_A = in^{A\ddagger}"] & A \ddagger \textcircled{$\ddagger$}.B \arrow[r, "Id \ddagger \llparenthesis \beta_B \rrparenthesis_{B\ddagger}"] \arrow[d, "f \ddagger Id"] & A \ddagger K.B \arrow[d, "f \ddagger Id"] \\
 & B \ddagger \textcircled{$\ddagger$}.B \arrow[r, "Id \ddagger \llparenthesis \beta_B \rrparenthesis_{B\ddagger}"] \arrow[d, "in^{B\ddagger}"] & B \ddagger K.B \arrow[d, "\beta_B"] \\
\textcircled{$\ddagger$}.A \arrow[r, "\textcircled{$\ddagger$}.f"] & \textcircled{$\ddagger$}.B \arrow[r, "\llparenthesis \beta_B \rrparenthesis_{B\ddagger}"] & K.B
\end{tikzcd}$$

The top-right square commutes trivially, the bottom-right because it is a catamorphism diagram. Then we have

$$\begin{array}{ll}
 & K.f \circ h_A \\
= & \{\ \text{first fusion calculation}\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \llparenthesis \beta_B \circ (f \ddagger Id) \rrparenthesis_{A\ddagger} \\
= & \quad \{ \text{second fusion calculation} \} \\
 & h_B \circ \textcircled{\ddagger}.f
\end{array}
$$

So h is natural as required.

We must still check that h is an $(Id_{\mathbb{C}} \dot{\ddagger})$-algebra homomorphism, that is,

$$h \circ \alpha = \beta \circ Id_{\mathbb{C}} \dot{\ddagger}\, h$$

We calculate for each A in $\mathbb{C}$:

$$
\begin{array}{ll}
 & (h \circ \alpha)_A \\
= & \\
 & h_A \circ \alpha_A \\
= & \quad \{ \text{definitions of } h \text{ and } \alpha \} \\
 & \llparenthesis \beta_A \rrparenthesis_{A\ddagger} \circ in^{\ddagger} \\
= & \quad \{ \text{catamorphism} \} \\
 & \beta_A \circ (Id_A \ddagger \llparenthesis \beta_A \rrparenthesis_{A\ddagger}) \\
= & \quad \{ \text{definition of } h \} \\
 & \beta_A \circ (Id_A \ddagger h_A) \\
= & \\
 & (\beta \circ Id_{\mathbb{C}} \dot{\ddagger}\, h)_A
\end{array}
$$

We have proved the existence of h — we must next prove the uniqueness. Let $k : \textcircled{\ddagger} \underset{\bullet}{\longrightarrow} K$ be any $(Id_{\mathbb{C}} \dot{\ddagger})$-algebra homomorphism between α and β, so:

$$\begin{tikzcd}
Id_{\mathbb{C}} \dot{\ddagger} \textcircled{$\ddagger$} \arrow[r, "Id \dot{\ddagger} k"] \arrow[d, "\alpha"] & Id_{\mathbb{C}} \dot{\ddagger} K \arrow[d, "\beta"] \\
\textcircled{$\ddagger$} \arrow[r, "k"] & K
\end{tikzcd}$$

Then for every A in $\mathbb{C}$,

$$(k \circ \alpha)_A = (\beta \circ Id_{\mathbb{C}} \dot{\ddagger} k)_A$$

$\equiv$

$$k_A \circ \alpha_A = \beta_A \circ (Id_{\mathbb{C}} \dot{\ddagger} k)_A$$

$\equiv$ { definition of α }

$$k_A \circ in^{A\ddagger} = \beta_A \circ (Id_A \ddagger k_A)$$

$\equiv$ { catamorphism characterization }

$$k_A = (\!| \beta_A |\!)_{A\ddagger}$$

$\equiv$ { definition of h }

$$k_A = h_A$$

So k must equal h, and so k is unique.

Therefore α is an initial algebra, and we can justify calling it $in^{Id_{\mathbb{C}}\ddagger}$:

$$in^{Id_{\mathbb{C}}\ddagger} \stackrel{def}{=} \alpha$$

A.2. Lifting functors

Lemma A.1 (lifting preserves cocontinuity): If $\mathbb{C}$ is $\mathbb{J}$-cocomplete and $F : \mathbb{C} \to \mathbb{D}$ is $\mathbb{J}$-cocontinuous for some small category $\mathbb{J}$, then so is the lifting of $F, \dot{F} : \mathbb{C}^{\mathbb{B}} \to \mathbb{D}^{\mathbb{B}}$, for any small category $\mathbb{B}$.

Proof: We must show that $\dot{F}$ preserves $\mathbb{J}$-colimits. Let $\mu : D \rhd U$ be a colimiting cocone under some diagram $D : \mathbb{J} \to \mathbb{C}^{\mathbb{B}}$. We must show that the cocone

$$\dot{F}\mu : \dot{F}D \rhd \dot{F}.U$$

is colimiting under the diagram

$$\mathbb{J} \xrightarrow{D} \mathbb{C}^{\mathbb{B}} \xrightarrow{\dot{F}} \mathbb{D}^{\mathbb{B}}$$

That is, given any other cocone

$$\alpha : \dot{F}D \rhd G$$

under $\dot{F}D$ then there is a unique cocone morphism from $\dot{F}\mu$ to α. That is, a unique mediating morphism $h : \dot{F}.U \to G$ in $\mathbb{D}^{\mathbb{B}}$ such that for every object j in $\mathbb{J}$,

$$\begin{tikzcd}
& \dot{F}.U \arrow[dd, "h"] \\
\dot{F}D.j \arrow[ur, "(\dot{F}\mu)_j"] \arrow[dr, "\alpha_j"'] & \\
& G
\end{tikzcd} \tag{A.2.1}$$

Let α be given. Then we must construct the mediating morphism $h : \dot{F}.U \to G$. Note that h must be an arrow in $\mathbb{D}^{\mathbb{B}}$, and therefore a natural transformation. So for each object B in $\mathbb{B}$, we must find the component

$$h_B : (\dot{F}.U).B \to G.B$$

But notice that

$$
\begin{array}{ll}
 & (\dot{F}.U).B \\
= & \qquad \{ \text{ definition of } (@B) \} \\
 & (@B).(\dot{F}.U) \\
= & \qquad \{ \text{ functor composition } \} \\
 & (@B)\dot{F}.U \\
= & \qquad \{ \text{ lemma 3.1 } \} \\
 & F(@B).U
\end{array}
$$

So we are actually looking for an arrow of type

$$h_B : F(@B).U \to G.B$$

We know by corollary 3.1 that $(@B)$ is cocontinuous, and F is $\mathbb{J}$-cocontinuous by assumption, so their composite,

$$\mathbb{C}^{\mathbb{B}} \xrightarrow{(@B)} \mathbb{C} \xrightarrow{F} \mathbb{D}$$

is also $\mathbb{J}$-cocontinuous. Then, applying $F(@B)$ to our original colimit μ yields another colimiting cocone

$$F(@B)\mu : F(@B)D \rhd F(@B).U$$

this time under the diagram

$$\mathbb{J} \xrightarrow{D} \mathbb{C}^{\mathbb{B}} \xrightarrow{(@B)} \mathbb{C} \xrightarrow{F} \mathbb{D}$$

To make use of this colimit to find h_B, we must use α and G to construct another cocone under $F(@B)D$. The cocone we need is

$$(@B)\alpha : (@B)\dot{F}D \rhd (@B).G$$

which, using lemma 3.1 and the definition of $(@B)$, we can rewrite the type of as

$$(@B)\alpha : F(@B)D \rhd G.B$$

Then, because $F(@B)\mu$ is colimiting under $F(@B)D$, we get a mediating morphism

$$h_B : F(@B).U \rightarrow G.B$$

such that for each object j in $\mathbb{J}$,

$$h_B \circ (F(@B)\mu)_j = ((@B)\alpha)_j$$

or equivalently

$$h_B \circ F.\left(\mu_j\right)_B = \left(\alpha_j\right)_B \qquad \text{(A.2.2)}$$

Thus we can construct the components of h for each object B in $\mathbb{B}$. It remains to show that the h so defined is, first of all, a natural transformation and secondly, the mediating morphism from $\dot{F}\mu$ to α.

Let us prove that h is a natural transformation. Given any $f : B \rightarrow C$ in $\mathbb{B}$, we must show

$$\begin{tikzcd}
(\dot{F}.U).B \arrow[r, "h_B"] \arrow[d, "(\dot{F}.U).f"'] & G.B \arrow[d, "G.f"] \\
(\dot{F}.U).C \arrow[r, "h_C"'] & G.C
\end{tikzcd}$$

To do this, we construct another cocone

$$\beta : F(@B)D \rhd G.C$$

by defining for each object j in $\mathbb{J}$,

$$\beta_j \overset{def}{=} F(@B)D.j \xrightarrow{\left(\alpha_j\right)_B} G.B \xrightarrow{G.f} G.C$$

This can easily be shown to be a cocone by using the fact that α is a cocone. Then, because $F(@B)\mu$ is colimiting under $F(@B)D$, we get a new mediating morphism

$$k : F(@B).U \to G.C$$

that uniquely satisfies

$$\begin{tikzcd} & F(@B).U \arrow[dd, "k"] \\ F(@B)D.j \arrow[ur, "(F(@B)\mu)_j"] \arrow[dr, "\beta_j"'] & \\ & G.C \end{tikzcd} \tag{A.2.3}$$

for each object j in $\mathbb{J}$. But $G.f$ also satisfies equation A.2.3:

$$\begin{array}{ll} & \beta_j \\ = & \quad \{ \text{definition of } \beta \} \\ & G.f \circ \left(\alpha_j\right)_B \\ = & \quad \{ \text{equation A.2.2} \} \\ & G.f \circ h_B \circ F.\left(\mu_j\right)_B \\ = & \quad \{ \text{definition of } (@B) \} \\ & G.f \circ h_B \circ (F(@B)\mu)_j \end{array}$$

for any object j in $\mathbb{J}$, and therefore, by the uniqueness of k,

$$k = G.f \circ h_B \tag{A.2.4}$$

which is one direction in our naturality square. Next we prove that the other direction in our naturality square also equals k. For any object j in $\mathbb{J}$,

$$\beta_j$$

$=$ { definition of β }

$$G.f \circ \left(\alpha_j \right)_B$$

$=$ { naturality of α_j }

$$\left(\alpha_j \right)_C \circ F.(D.j).f$$

$=$ { equation A.2.2 }

$$h_C \circ F.\left(\mu_j \right)_C \circ F.(D.j).f$$

$=$ { F is a functor }

$$h_C \circ F.\left(\left(\mu_j \right)_C \circ (D.j).f \right)$$

$=$ { naturality of μ_j }

$$h_C \circ F.\left(U.f \circ \left(\mu_j \right)_B \right)$$

$=$ { F is a functor }

$$h_C \circ F.U.f \circ F.\left(\mu_j \right)_B$$

$=$ { definitions of $\dot{F}$ and $(@B)$ }

$$h_C \circ (\dot{F}.U).f \circ (F(@B)\mu)_j$$

and so $h_C \circ (\dot{F}.U).f$ also satisfies equation A.2.3, and again by uniqueness of k we have

$$h_C \circ (\dot{F}.U).f$$

$=$ { uniqueness of k }

$$k$$

$=$ { equation A.2.4 }

$$G.f \circ h_B$$

Thus our naturality square commutes and h is a natural transformation.

Finally we prove the uniqueness of the cocone morphism h. Let $g : \dot{F}.U \underset{\cdot}{\longrightarrow} G$ be another cocone morphism from $\dot{F}\mu$ to α, so for any object j in $\mathbb{J}$,

(A.2.5)

$$\begin{tikzcd}
 & \dot{F}.U \arrow[dd, "g"] \\
\dot{F}D.j \arrow[ru, "(\dot{F}\mu)_j"] \arrow[rd, "\alpha_j"'] & \\
 & G
\end{tikzcd}$$

Then we prove that g equals h. We do this componentwise, so for each object B in $\mathbb{B}$,

$$\begin{aligned}
& \left(\alpha_j\right)_B \\
= & \qquad \{\, \text{equation A.2.5} \,\} \\
& \left(g \circ (\dot{F}\mu)_j\right)_B \\
= & \qquad \{\, \text{pointwise composition of natural transformation} \,\} \\
& g_B \circ \left((\dot{F}\mu)_j\right)_B \\
= & \qquad \{\, \text{definition of } (@B) \,\} \\
& g_B \circ (@B)\dot{F}.\mu_j \\
= & \qquad \{\, \text{lemma 3.1} \,\} \\
& g_B \circ F(@B).\mu_j \\
= & \qquad \{\, \text{definition of } (@B) \,\} \\
& g_B \circ F.\left(\mu_j\right)_B
\end{aligned}$$

but h_B is the unique solution of this for all j in $\mathbb{J}$, so g_B equals h_B, and therefore $g = h$.

A.3. Order-enriched categories

Lemma A.2 (locally ω-continuous functors preserve O-colimits): Let $F : \mathbb{C} \to \mathbb{D}$ be locally ω-continuous and let $D : \omega \to \mathbb{C}_E$ be an ω-diagram in $\mathbb{C}_E$ with an **O**-colimit $\mu : D \vartriangleright A$ for some A in $\mathbb{C}_E$. Then $F_E\mu : F_E D \vartriangleright F_E.A$ is an **O**-colimit of the diagram

$$\omega \xrightarrow{D} \mathbb{C}_E \xrightarrow{F_E} \mathbb{D}_E$$

where F_E is the restriction of F to embeddings.

Proof: For each i in ω, $\mu_i : D.i \to A$ is an arrow in $\mathbb{C}_E$. In particular, this means that μ_i is an embedding with retraction $\left(\mu_i\right)^R : A \to D.i$. We know that locally ω-continuous functions are locally monotonic, and therefore preserve projection pairs. This gives us that

$$\left(F_E.\mu_i\right)^R = F_E.\left(\mu_i\right)^R \tag{A.3.1}$$

and we note that

$$F_E.\left(\mu_i \circ \left(\mu_i\right)^R\right)$$

$=$ $\{\, F_E \text{ is a functor} \,\}$

$$F_E.\mu_i \circ F_E.\left(\mu_i\right)^R$$

$=$ $\{\, \text{equation A.3.1} \,\}$

$$F_E.\mu_i \circ \left(F_E.\mu_i\right)^R$$

$=$ $\{\, \text{composition of functors and natural transformations} \,\}$

$$\left(F_E\mu\right)_i \circ \left(\left(F_E\mu\right)_i\right)^R$$

So

$$F_E.\Big(\mu_i \circ (\mu_i)^R\Big) = (F_E\mu)_i \circ \Big((F_E\mu)_i\Big)^R \tag{A.3.2}$$

for any i in ω. We also note that μ is an **O**-colimit, which implies that

$$\left\{ A \xrightarrow{(\mu_i)^R} D.i \xrightarrow{\mu_i} A \right\}_{i\in\omega}$$

is ascending, or, in other words, for every i in ω,

$$\mu_i \circ (\mu_i)^R \sqsubseteq_{\mathbb{C}} \mu_{i+1} \circ (\mu_{i+1})^R \tag{A.3.3}$$

We must show that

$$\left\{ F_E.A \xrightarrow{\left((F_E\mu)_i\right)^R} F_E D.i \xrightarrow{(F_E\mu)_i} F_E.A \right\}_{i\in\omega}$$

is an ascending ω-chain with least upper bound $Id_{F_E.A}$. We first show that it is ascending. For every i in ω,

$$(F_E\mu)_i \circ \Big((F_E\mu)_i\Big)^R$$

$=$ { equation A.3.2 }

$$F_E.\Big(\mu_i \circ (\mu_i)^R\Big)$$

$\sqsubseteq_{\mathbb{D}}$ { F_E is locally monotonic, equation A.3.3 }

$$F_E.\Big(\mu_{i+1} \circ (\mu_{i+1})^R\Big)$$

$=$ { equation A.3.2 }

$$(F_E\mu)_{i+1} \circ \Big((F_E\mu)_{i+1}\Big)^R$$

Now we show that the least upper bound is equal to $Id_{F_E.A}$:

$$\bigsqcup\left\{ \left(F_E\mu\right)_i \circ \left(\left(F_E\mu\right)_i\right)^R \right\}_{i\in\omega}$$

$=$ { equation A.3.2 }

$$\bigsqcup\left\{ F_E.\left(\mu_i \circ \left(\mu_i\right)^R\right) \right\}_{i\in\omega}$$

$=$ { F_E is locally ω-continuous }

$$F_E.\bigsqcup\left\{ \mu_i \circ \left(\mu_i\right)^R \right\}_{i\in\omega}$$

$=$ { μ is an **O**-colimit }

$$F_E.Id_A$$

$=$ { F_E a functor }

$$Id_{F_E.A}$$

A.4. Order-enriching functor categories

Lemma A.3 (a functor category is order-enriched if its target category is): If $\mathbb{D}$ is an **O**-category and $\mathbb{C}$ is a small category, then $\mathbb{D}^{\mathbb{C}}$ is also an **O**-category — the ordering on the hom-sets $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$ of $\mathbb{D}^{\mathbb{C}}$ being defined componentwise from the ordering on the hom-sets of $\mathbb{D}$:

$$\alpha \sqsubseteq_{\mathbb{D}^{\mathbb{C}}} \beta \;\equiv\; \forall A \in \mathbb{C}, \alpha_A \sqsubseteq_{\mathbb{D}} \beta_A$$

for any objects F and G in $\mathbb{D}^{\mathbb{C}}$. Furthermore, least upper bounds of ascending ω-chains in the hom-sets of $\mathbb{D}^{\mathbb{C}}$ can be computed pointwise:

$$\left(\bigsqcup\left\{\alpha^i : F \underset{\bullet}{\to} G\right\}_{i\in\omega}\right)_A = \bigsqcup\left\{\alpha^i_A : F.A \to G.A\right\}_{i\in\omega}$$

Proof: We first show that ordering defined above is a partial ordering. Reflexivity and transitivity can easily be shown using a pointwise argument, and we can do the same for antisymmetry: Suppose we have two natural transformations α and β in $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$ such that $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \alpha$, then we must prove that $\alpha = \beta$. For any object A in $\mathbb{C}$, we know that $\alpha_A \sqsubseteq \beta_A$ and $\beta_A \sqsubseteq \alpha_A$ with respect to the ordering on $hom_D(F.A, G.A)$. Then, by antisymmetry of this ordering we know that $\alpha_A = \beta_A$, and therefore $\alpha = \beta$.

Next we show the hom-sets to be ω-complete. Let $\left\{ \alpha^i : F \underset{\bullet}{\longrightarrow} G \right\}_{i\in\omega}$ be an ascending ω-chain in $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$. We must show that this chain has a least upper bound in $hom_{\mathbb{D}^{\mathbb{C}}}(F, G)$.

For each A in $\mathbb{C}$, the set $\left\{ \alpha^i_A : F.A \to G.A \right\}_{i\in\omega}$ is an ascending ω-chain in $hom_{\mathbb{D}}(F.A, G.A)$ and therefore has a least upper bound because $hom_{\mathbb{D}}(F.A, G.A)$ is ω-complete (because $\mathbb{D}$ is an **O**-category). This suggests that the least upper bound of $\left\{ \alpha^i : F \to G \right\}_{i\in\omega}$ can be computed pointwise to give a natural transformation $\lambda : F \underset{\bullet}{\longrightarrow} G$ where

$$\lambda_A \overset{def}{=} \bigsqcup \left\{ \alpha^i_A \right\}_{i\in\omega} \tag{A.4.1}$$

But can we be sure that λ is actually a natural transformation and that it is the least upper bound of $\left\{ \alpha^i \right\}_{i\in\omega}$?

First we show λ to be natural. We want that for any $f : A \to B$ in $\mathbb{C}$, that

$$\begin{tikzcd}
F.A \arrow[r, "\lambda_A"] \arrow[d, "F.f"] & G.A \arrow[d, "G.f"] \\
F.B \arrow[r, "\lambda_B"] & G.B
\end{tikzcd}$$

We calculate

$$
\begin{aligned}
& \lambda_B \circ F.f = G.f \circ \lambda_A \\
\equiv \quad & \{\text{ definition of } \lambda \text{ (A.4.1) }\} \\
& \sqcup\left\{ \alpha^i_B \right\}_{i\in\omega} \circ F.f = G.f \circ \sqcup\left\{ \alpha^i_A \right\}_{i\in\omega} \\
\equiv \quad & \{\text{ composition is } \omega\text{-continuous }\} \\
& \sqcup\left\{ \alpha^i_B \circ F.f \right\}_{i\in\omega} = \sqcup\left\{ G.f \circ \alpha^i_A \right\}_{i\in\omega}
\end{aligned}
$$

But notice that for any i in ω,

$$\begin{tikzcd}
F.A \arrow[r, "\alpha^i_A"] \arrow[d, "F.f"] & G.A \arrow[d, "G.f"] \\
F.B \arrow[r, "\alpha^i_B"] & G.B
\end{tikzcd}$$

by the naturality of α^i. So the two ω-chains are, in fact, identical and consequently so are their least upper bounds. Thus we have shown λ to be natural.

Next we show that λ is actually the least upper bound of $\left\{\alpha^i\right\}_{i\in\omega}$. Let $\gamma : F \xrightarrow{\cdot} G$ be any other upper bound. Then for every A in $\mathbb{C}$, $\gamma_A : F.A \rightarrow G.A$ is an upper bound for $\left\{ \alpha^i_A \right\}_{i\in\omega}$. But we know by definition that λ_A is the *least* upper bound of this chain, so it must be the case that $\lambda_A \sqsubseteq \gamma_A$. Because this is true for every A in $\mathbb{C}$, it follows that $\lambda \sqsubseteq \gamma$ by the definition of the ordering. Thus λ is the least upper bound. In particular, this gives us:

$$\left(\sqcup\left\{\alpha^i\right\}_{i\in\omega} \right)_A = \sqcup\left\{ \alpha^i_A \right\}_{i\in\omega} \tag{A.4.2}$$

It remains to show that composition of natural transformations is ω-continuous in both arguments. Given any three objects F, G and H in $\mathbb{D}^{\mathbb{C}}$, consider the composition operator:

$$(\circ) : hom_{\mathbb{D}^{\mathbb{C}}}(G, H) \times hom_{\mathbb{D}^{\mathbb{C}}}(F, G) \to hom_{\mathbb{D}^{\mathbb{C}}}(F, H)$$

Then, for any natural transformations $\beta : F \mathrel{\dot{\to}} G$ and $\gamma : G \mathrel{\dot{\to}} H$, we must show that the sectioned composition operators:

$$(\gamma\circ) : hom_{\mathbb{D}^{\mathbb{C}}}(F, G) \to hom_{\mathbb{D}^{\mathbb{C}}}(F, H)$$

and

$$(\circ\beta) : hom_{\mathbb{D}^{\mathbb{C}}}(G, H) \to hom_{\mathbb{D}^{\mathbb{C}}}(F, H)$$

are both ω-continuous. Let $\left\{ \alpha^i \right\}_{i\in\omega}$ be an ascending ω-chain as before. We have shown that it has a least upper bound; we must now show that $(\gamma\circ)$ *preserves* that least upper bound. That is,

$$\gamma \circ \bigsqcup\left\{ \alpha^i \right\}_{i\in\omega} = \bigsqcup\left\{ \gamma \circ \alpha^i \right\}_{i\in\omega}$$

We do this pointwise, for each A in $\mathbb{C}$:

$$\left(\gamma \circ \bigsqcup\left\{ \alpha^i \right\}_{i\in\omega} \right)_A$$

$=$ { composition of natural transformations }

$$\gamma_A \circ \left(\bigsqcup\left\{ \alpha^i \right\}_{i\in\omega} \right)_A$$

$=$ { least upper bounds are computed pointwise (A.4.1) }

$$\gamma_A \circ \bigsqcup\left\{ \alpha^i_A \right\}_{i\in\omega}$$

$=$ { composition is ω-continuous in $\mathbb{D}$ }

$$\bigsqcup\left\{ \gamma_A \circ \alpha^i_A \right\}_{i\in\omega}$$

$=$ { composition of natural transformations }

$$\bigsqcup\left\{ \gamma \circ \alpha^i{}_A \right\}_{i\in\omega}$$

$=$ { least upper bounds are computed pointwise (A.4.1) }

$$\left(\sqcup \left\{ \gamma \circ \alpha^{i} \right\}_{i \in \omega} \right)_{A}$$

The proof for $(\circ\beta)$ is similar.

Lemma A.4 (If $\mathbb{D}$ is an O$\perp$-category, then so is $(\mathbb{D}_{\perp})^{\mathbb{C}}$): If $\mathbb{D}$ is an **O**$\perp$-category then for any small category $\mathbb{C}$, the functor category $(\mathbb{D}_{\perp})^{\mathbb{C}}$ is also an **O**$\perp$-category. For any functors F and G, the least element of the hom-set $hom_{(\mathbb{D}_{\perp})^{\mathbb{C}}}(F, G)$ has components:

$$\left(\perp_{F \to G} \right)_{A} \overset{def}{=} \perp_{F.A \to G.A}$$

for each A in $\mathbb{C}$.

Proof: If $\mathbb{D}$ is an **O**$\perp$ category then so is the subcategory $\mathbb{D}_{\perp}$ containing only the strict maps. By theorem NumberOfoe-funcat-lem, we see that we define an ordering on hom-sets that makes $(\mathbb{D}_{\perp})^{\mathbb{C}}$ an **O**-category. We must show $(\mathbb{D}_{\perp})^{\mathbb{C}}$ to be an **O**$\perp$-category, that is, that each hom-set has a least element that is a post-zero of composition.

We start by finding the least element of $hom_{(\mathbb{D}_{\perp})^{\mathbb{C}}}(F, G)$ for any F and G. The obvious choice for the least element is to consider the natural transformation whose components are all least elements of their respective hom-sets in $\mathbb{D}_{\perp}$. That is, for each A in $\mathbb{C}$, define:

$$(\perp_{F \to G})_{A} \overset{def}{=} \perp_{F.A \to G.A}$$

We must show that this really is a natural transformation, and secondly that it is the least element with respect to the ordering on the hom-set. For naturality we calculate for any $f : A \to B$ in $\mathbb{C}$:

$$(\perp_{F \to G})_{B} \circ F.f$$

$$
\begin{array}{ll}
= & \{\text{ definition of } \bot_{F\to G} \} \\
& \bot_{F.B\to G.B} \circ F.f \\
= & \{ \bot \text{ is post-zero of composition in } \mathbb{D}_\bot \} \\
& \bot_{F.A\to G.B} \\
= & \{\text{ G.f must be strict because it is in } \mathbb{D}_\bot \} \\
& G.f \circ \bot_{F.A\to G.A} \\
= & \{\text{ definition of } \bot_{F\to G} \} \\
& G.f \circ (\bot_{F\to G})_A
\end{array}
$$

We see in this calculation why we must restrict the proof to the strict arrows of $\mathbb{D}$ — we need $G.f$ to be strict.

Now we must show that $\bot_{F\to G}$ is the *least* element. Let $\alpha : F \to G$ be any other arrow in $(\mathbb{D}_\bot)^{\mathbb{C}}$, then for each A in $\mathbb{C}$ we know that

$$
\begin{array}{ll}
& (\bot_{F\to G})_A \\
= & \{\text{ definition of } \bot_{F\to G} \} \\
& \bot_{F.A\to G.A} \\
\sqsubseteq & \{ \bot \text{ is the least element } \} \\
& \alpha_A
\end{array}
$$

and therefore, by definition of the ordering on $hom_{(\mathbb{D}_\bot)^{\mathbb{C}}}(F,G)$, we know that $\bot_{F\to G} \sqsubseteq \alpha$.

Lastly, we must show that $\bot_{F\to G}$ is a post-zero of composition. Let $\alpha : H \to F$ be another arrow in $(\mathbb{D}_\bot)^{\mathbb{C}}$, then for any A in $\mathbb{C}$,

$$
\begin{array}{ll}
& (\bot_{F\to G} \circ \alpha)_A \\
= & \{\text{ composition of natural transformations } \} \\
& (\bot_{F\to G})_A \circ \alpha_A \\
= & \{\text{ definition of } \bot_{F\to G} \} \\
& \bot_{F.A\to G.A} \circ \alpha_A
\end{array}
$$

$=$ { $\perp$ is post-zero of composition in $\mathbb{D}_\perp$ }

$\perp_{H.A \to G.A}$

$=$ { definition of $\perp_{H \to G}$ }

$(\perp_{H \to G})_A$

And we are finished.

Lemma A.5 (functor categories are localized if their target categories are): Let $\mathbb{D}$ be a localized **O**-category and $\mathbb{C}$ any small category. Then $\mathbb{D}^\mathbb{C}$ is a localized category.

Proof: By lemma 3.10 we know that $\mathbb{D}^\mathbb{C}$ is an **O**-category. Let $D : \omega \to \left(\mathbb{D}^\mathbb{C}\right)_{PR}$ be an ω-diagram and $\alpha : D \mathrel{\triangleright\!\!\!\rightarrow} F$ a cocone under it. We must find an object G in $\mathbb{D}^\mathbb{C}$ and a projection pair $h = \left(h^L, h^R\right)$ from G to F, that is, natural transformations,

$$h^L : G \underset{\bullet}{\longrightarrow} F \qquad\qquad h^R : F \underset{\bullet}{\longrightarrow} G$$

that form a projection pair.

For each A in $\mathbb{C}$, consider the diagram

$$\omega \xrightarrow{D} \left(\mathbb{D}^\mathbb{C}\right)_{PR} \xrightarrow{(@A)_{PR}} \mathbb{D}_{PR}$$

We know that $(@A)_{PR}\alpha : (@A)_{PR} \mathrel{\triangleright\!\!\!\rightarrow} F.A$ is a cocone under $(@A)_{PR}D$. Then, because $\mathbb{D}$ is localized, there exists some object G_A in $\mathbb{C}$ and a projection pair $h_A = (h^L_A, h^R_A)$ from G_A to $F.A$ satisfying

$$h^L_A \circ h^R_A = \bigsqcup \left\{ F.A \xrightarrow{\left(\alpha_i\right)^R_A} (D.i).A \xrightarrow{\left(\alpha_i\right)^L_A} F.A \right\}_{i \in \omega} \tag{A.4.3}$$

We can do this for any A in $\mathbb{C}$, and use the resulting G_A's and h_A's to construct our required functor G:

$$\begin{tikzcd}[row sep=small] G : \mathbb{C} \arrow[r] & \mathbb{D} \\ A \arrow[d, "f"'] & G_A \arrow[d, "h^R_B \circ F.f \circ h^L_A"] \\ B & G_B \end{tikzcd}$$

$$\mapsto$$

We can check that G is a functor, that is, that it preserves identities:

$$
\begin{array}{ll}
& G.Id_A \\
= & \quad \{ \text{definition of } G \} \\
& h^R_B \circ F.Id_A \circ h^L_A \\
= & \quad \{ F \text{ is a functor} \} \\
& h^R_B \circ Id_{F.A} \circ h^L_A \\
= & \quad \{ \text{identity} \} \\
& h^R_B \circ h^L_A \\
= & \quad \{ \text{projection pair} \} \\
& Id_{G.A}
\end{array}
$$

To prove preservation of composite arrows is a little more involved. Let us first note that for any $k : A \to B$ in $\mathbb{C}$,

$$
\begin{array}{ll}
& h^L_B \circ h^R_B \circ F.k \\
= & \quad \{ \text{equation A.4.3} \} \\
& \bigsqcup \left\{ \left(\alpha_i \right)^L_B \circ \left(\alpha_i \right)^R_B \right\}_{i \in \omega} \circ F.k \\
= & \quad \{ \omega\text{-continuity of composition} \} \\
& \bigsqcup \left\{ \left(\alpha_i \right)^L_B \circ \left(\alpha_i \right)^R_B \circ F.k \right\}_{i \in \omega} \\
= & \quad \{ \text{naturality of } \left(\alpha_i \right)^L \text{ and } \left(\alpha_i \right)^R \} \\
& \bigsqcup \left\{ F.k \circ \left(\alpha_i \right)^L_A \circ \left(\alpha_i \right)^R_A \right\}_{i \in \omega} \\
= & \quad \{ \omega\text{-continuity of composition} \} \\
& F.k \circ \bigsqcup \left\{ \left(\alpha_i \right)^L_A \circ \left(\alpha_i \right)^R_A \right\}_{i \in \omega}
\end{array}
$$

$=$ { equation A.4.3 }

$F.k \circ h^L_A \circ h^R_A$

So we have

$$h^L_B \circ h^R_B \circ F.k = F.k \circ h^L_A \circ h^R_A \tag{A.4.4}$$

Then the proof that G preserves composition is simple — for any arrows $f : A \to B$ and $g : B \to C$ in $\mathbb{C}$,

$G.g \circ G.f$

$=$ { definition of G }

$h^R_C \circ F.g \circ h^L_B \circ h^R_B \circ F.f \circ h^L_A$

$=$ { equation A.4.4 }

$h^R_C \circ F.g \circ F.f \circ h^L_A \circ h^R_A \circ h^L_A$

$=$ { projection pair }

$h^R_C \circ F.g \circ F.f \circ h^L_A \circ Id_{G.A}$

$=$ { F a functor, identity }

$h^R_C \circ F.(g \circ f) \circ h^L_A$

$=$ { definition of G }

$G.(g \circ f)$

So G is a functor. We can also prove that the h^L_A's and h^R_A's are natural. Let $f : A \to B$ be any arrow in $\mathbb{D}$. Then we calculate

$h^L_B \circ G.f$

$=$ { definition of G }

$h^L_B \circ h^R_B \circ F.f \circ h^L_A$

$=$ { equation A.4.4 }

$F.f \circ h^L_A \circ h^R_A \circ h^L_A$

$=$ { projection pair }

$$
\begin{aligned}
& F.f \circ h^L_A \circ Id_{G.A} \\
= & \quad \{\text{ identity }\} \\
& F.f \circ h^L_A
\end{aligned}
$$

So h^L is a natural transformation. The proof that h^R is natural is similar.

Because $\left(h^L_A, h^R_A\right)$ is a projection pair for every A in $\mathbb{C}$, we know by lemma 3.12 that $\left(h^L, h^R\right)$ is a projection pair in $\mathbb{D}^{\mathbb{C}}$.

The only remaining thing to prove is that

$$h^L \circ h^R = \bigsqcup\left\{\left(\alpha_i\right)^L \circ \left(\alpha_i\right)^R\right\}_{i\in\omega}$$

and this is a simple pointwise calculation — for every A in $\mathbb{C}$,

$$
\begin{aligned}
& \left(h^L \circ h^R\right)_A \\
= & \quad \{\text{ pointwise composition of natural transformations }\} \\
& h^L_A \circ h^R_A \\
= & \quad \{\text{ equation A.4.3 }\} \\
& \bigsqcup\left\{\left(\alpha_i\right)^L_A \circ \left(\alpha_i\right)^R_A\right\}_{i\in\omega} \\
= & \quad \{\text{ pointwise composition of natural transformations }\} \\
& \bigsqcup\left\{\left(\left(\alpha_i\right)^L \circ \left(\alpha_i\right)^R\right)_A\right\}_{i\in\omega} \\
= & \quad \{\text{ least upper bounds are computed pointwise }\} \\
& \left(\bigsqcup\left\{\left(\alpha_i\right)^L \circ \left(\alpha_i\right)^R\right\}_{i\in\omega}\right)_A
\end{aligned}
$$

So we have proved that $\mathbb{D}^{\mathbb{C}}$ is localized.

A.5. Order-enriching algebra families

Lemma A.6 (injective re-indexings are continuous and cocontinuous): Let I be a set and $f : I \to I$ an injective function. Consider I as a *discrete category* and f as an endofunctor on it. Then

$$(\circ f) : \mathbb{C}^I \longrightarrow \mathbb{C}^I \tag{A.5.1}$$

$$\begin{tikzcd} X \arrow[d, "\alpha"'] \arrow[rrd, phantom, "\mapsto"] & & Xf \arrow[d, "\alpha f"] \\ Y & & Yf \end{tikzcd}$$

is both continuous and cocontinuous.

Proof: We show $(\circ f)$ to be continuous — the proof that $(\circ f)$ is cocontinuous follows the same structure.

Let $D : \mathbb{J} \to \mathbb{C}^I$ be some $\mathbb{J}$-diagram in $\mathbb{C}^I$ with limiting cone $\lambda : Lim(D) \lhd D$. We can construct a new diagram, $(\circ f)D : \mathbb{J} \to \mathbb{C}^I$, and a cone over it, $(\circ f)\lambda : (\circ f).Lim(D) \lhd (\circ f)D$. We would like to show that this is a *limiting* cone, that is, that $(\circ f)$ preserves limits.

Let $\beta : U \lhd (\circ f)D$ be another cone over $(\circ f)D$. We need to construct a *mediating morphism* $h : U \to (\circ f).Lim(D)$. The trick is to construct a new cone $\gamma : W \lhd D$ over D by extending β.

Because $f : I \to I$ is injective, there is a corresponding isomorphism:

$$g : I \cong Im(f)$$

where g is just f with restricted codomain. In particular,

$$\forall i \in I, f(i) = g(i) \tag{A.5.2}$$

If we consider $(\circ f)$ to be re-indexing the family, then g^{-1} will allow us to undo that indexing:

$$\Big((\circ f) D.\psi \Big)_{g^{-1}(i)}$$

$=$ { definition of $(\circ f)$ }

$$\Big((D.\psi) \circ f \Big)_{g^{-1}(i)}$$

$=$ { write using indexing notation }

$$\Big(D.\psi \Big)_{f(g^{-1}(i))}$$

$=$ { equation A.5.2 }

$$\Big(D.\psi \Big)_{g(g^{-1}(i))}$$

$=$ { inverse }

$$\Big(D.\psi \Big)_{i}$$

So

$$\Big((\circ f) D.\psi \Big)_{g^{-1}(i)} = \Big(D.\psi \Big)_{i} \tag{A.5.3}$$

We want to construct a new cone $\gamma : W \lhd D$ that copies the structure of β as far as possible. Let us first consider how we might build W — the apex of the cone. W is an I-indexed family of $\mathbb{C}$-objects. We define it:

$$\forall i \in I,\, W_i \overset{def}{=} \begin{cases} U_{g^{-1}(i)} & if\ i \in Im(f) \\ Lim(D)_i & otherwise. \end{cases} \tag{A.5.4}$$

Next we construct the cone γ — for each j in $\mathbb{J}$ we need $\gamma_j : W \to D.j$. But γ_j must be an I-indexed family of $\mathbb{C}$-arrows, so we need for each i in I, $\gamma_{j_i} : W_i \to (D.j)_i$. When $i \in Im(f)$, this translates to

$$\gamma_{j_i} : U_{g^{-1}(i)} \to (D.j)_i$$

by definition A.5.4. But note that

$$(D.j)_i$$

$=$ { isomorphism }

$(D.j)_{g(g^{-1}(i))}$

$=$ { g is the restriction of f }

$(D.j)_{f(g^{-1}(i))}$

$=$ { composition of functors }

$((D.j)f)_{g^{-1}(i)}$

$=$ { definition of $(\circ f)$ (5.7.1) }

$((\circ f).D.j)_{g^{-1}(i)}$

$=$ { composition of functors }

$((\circ f)D.j)_{g^{-1}(i)}$

So, in fact, when i is in the image of f, γ_{j_i} must have type:

$$\gamma_{j_i} : U_{g^{-1}(i)} \to ((\circ f)D.j)_{g^{-1}(i)}$$

and a suitable candidate is $(\beta_j)_{g^{-1}(i)}$.

Note that the above calculation is valid not only for objects j, but also arrows $\psi : j \to j'$. That is,

$$\forall i \in Im(f), \forall \psi : j \to j' \in \mathbb{J}, (D.\psi)_i = ((\circ f)D.\psi)_{g^{-1}(i)} \qquad \text{(A.5.5)}$$

We will use this result later.

What about the case when $i \notin Im(f)$? Then we need γ_{j_i} to have type

$$\gamma_{j_i} : Lim(D)_i \to (D.j)_i$$

We have an obvious candidate — the component from the limiting cone, λ_{j_i}.

Combining the two cases, we define:

$$\forall i \in I, \gamma_{j_i} \overset{def}{=} \begin{cases} (\beta_j)_{g^{-1}(i)} & \textit{if } i \in Im(f) \\ \lambda_{j_i} & \textit{otherwise.} \end{cases} \tag{A.5.6}$$

Thus we have defined γ, but we have not yet shown it to be a cone over D. We must show, for any $\psi : j \to j'$ in $\mathbb{J}$, that

$$\begin{tikzcd} & W \arrow[dl, "\gamma_j"'] \arrow[dr, "\gamma_{j'}"] & \\ D.j \arrow[rr, "D.\psi"] & & D.j' \end{tikzcd}$$

We proceed as follows:

$$D.\psi \circ \gamma_j = \gamma_{j'}$$

$\equiv$ { equality of families }

$$\forall i \in I, (D.\psi \circ \gamma_j)_i = \gamma_{j'_i}$$

$\equiv$ { composition of families }

$$\forall i \in I, (D.\psi)_i \circ \gamma_{j_i} = \gamma_{j'_i}$$

Then it is necessary to split into the two cases where $i \in Im(f)$ and $i \notin Im(f)$:

Case $i \notin Im(f)$:

$$(D.\psi)_i \circ \gamma_{j_i}$$

$=$ { definition of γ A.5.6 }

$$(D.\psi)_i \circ \lambda_{j_i}$$

$=$ { composition of families }

$$(D.\psi \circ \lambda_j)_i$$

$$= \qquad \{\ \lambda \text{ is a cone over } D\ \}$$

$$(\lambda_{j'})_i$$

$$= \qquad \{\ \text{definition of } \gamma \text{ A.5.6}\ \}$$

$$(\gamma_{j'})_i$$

Case $i \in Im(f)$:

$$(D.\psi)_i \circ \gamma_{j_i}$$

$$= \qquad \{\ \text{definition of } \gamma \text{ A.5.6}\ \}$$

$$(D.\psi)_i \circ (\beta_j)_{g^{-1}(i)}$$

$$= \qquad \{\ \text{equation A.5.3}\ \}$$

$$((\circ f)D.\psi)_{g^{-1}(i)} \circ (\beta_j)_{g^{-1}(i)}$$

$$= \qquad \{\ \text{composition of families}\ \}$$

$$((\circ f)D.\psi \circ \beta_j)_{g^{-1}(i)}$$

$$= \qquad \{\ \beta \text{ is a cone over } (\circ f)D\ \}$$

$$(\beta_{j'})_{g^{-1}(i)}$$

$$= \qquad \{\ \text{definition of } \gamma \text{ A.5.6}\ \}$$

$$(\gamma_{j'})_i$$

Therefore γ is a cone over D. Because λ is a limiting cone, we get a mediating morphism:

$$k : W \rightarrow Lim(D)$$

which we will use to construct our mediating arrow $h : U \rightarrow (\circ f).Lim(D)$. But h is a family of arrows, so we need for each i in I,

$$h_i : U_i \rightarrow ((\circ f).Lim(D))_i$$

But note that

$$
\begin{aligned}
& ((\circ f).Lim(D))_i \\
= \quad & \{ \text{ definition of } (\circ f) \} \\
& (Lim(D)f)_i \\
= \quad & \{ \text{ composition of functors } \} \\
& Lim(D)_{f(i)} \\
= \quad & \{ \text{ equation A.5.2 } \} \\
& Lim(D)_{g(i)}
\end{aligned}
$$

and also that

$$
\begin{aligned}
& U_i \\
= \quad & \{ \text{ isomorphism } \} \\
& U_{g^{-1}(g(i))} \\
= \quad & \{ \text{ definition of } W \text{ A.5.4 } \} \\
& W_{g(i)}
\end{aligned}
$$

So in fact, h_i must have type $W_{g(i)} \to Lim(D)_{g(i)}$, and this is exactly the type of the $g(i)$-th component of k, so define:

$$h_i \overset{def}{=} k_{g(i)} \tag{A.5.7}$$

We must check that h so defined uniquely satisfies, for all j in $\mathbb{J}$:

(A.5.8)

$$\begin{tikzcd}
U \arrow[rr, "h"] \arrow[dr, "\beta_j"'] & & (\circ f).Lim(D) \arrow[dl, "((\circ f)\lambda)_j"] \\
& (\circ f)D.j &
\end{tikzcd}$$

We prove this pointwise for each i in I:

$$
\begin{aligned}
& (((\circ f)\lambda)_j \circ h)_i \\
= \quad & \{\text{ composition of families }\} \\
& ((\circ f)\lambda)_{j_i} \circ h_i \\
= \quad & \{\text{ composition of functors and natural transformations }\} \\
& ((\circ f).\lambda_j)_i \circ h_i \\
= \quad & \{\text{ definitions of } (\circ f) \text{ and } h \text{ }\} \\
& (\lambda_j)_{f(i)} \circ k_{g(i)} \\
= \quad & \{\text{ equation A.5.2 }\} \\
& (\lambda_j)_{g(i)} \circ k_{g(i)} \\
= \quad & \{\text{ composition of families }\} \\
& (\lambda_j \circ k)_{g(i)} \\
= \quad & \{\ k \text{ mediates between } \gamma \text{ and } \lambda \text{ }\} \\
& (\gamma_j)_{g(i)} \\
= \quad & \{\text{ definition of } \gamma \text{ A.5.6 }\} \\
& (\beta_j)_{g^{-1}(g(i))} \\
= \quad & \{\text{ isomorphism }\} \\
& \beta_{j_i}
\end{aligned}
$$

Next we must show that h is unique in this respect. Let $h' : U \to (\circ f).Lim(D)$ be another arrow satisfying

$$\forall j \in \mathbb{J}, ((\circ f)\lambda)_j \circ h' = \beta_j \tag{A.5.9}$$

Then we show that h' must equal h. From h' we can construct an arrow $k' : W \to Lim(D)$ by defining:

$$\forall i \in I, k'_i \overset{def}{=} \begin{cases} h'_{g^{-1}(i)} & if\ i \in Im(f) \\ k_i & otherwise. \end{cases} \tag{A.5.10}$$

It is easy to show that k' mediates between γ and λ, so by uniqueness we know that

$$k' = k \tag{A.5.11}$$

Then we argue that for each i in I,

h'_i

$=$ { isomorphism }

$h'_{g^{-1}(g(i))}$

$=$ { definition of k' A.5.10 }

$k'_{g(i)}$

$=$ { equation A.5.11 }

$k_{g(i)}$

$=$ { definition of h A.5.7 }

h_i

Therefore h is the unique mediating arrow, and $(\circ f)\lambda$ is a limit of $(\circ f)D$.

We have shown that $(\circ f)$ preserves arbitrary limits in $\mathbb{C}^I$, and is therefore continuous.

Appendix B. Basic category theory

For completeness, this appendix reviews some of the basic definitions that have been used throughout this thesis.

Definition B.1 (category): A *category* $\mathbb{C}$ consists of

(i) a class of *objects* $A, B, C, \ldots$;

(ii) for each pair of objects A and B, a *set* $hom_{\mathbb{C}}(A, B)$ called the set of *morphisms* or *arrows* from A to B;

(iii) for each triple of objects A, B and C, a partial map

$$\circ : hom_{\mathbb{C}}(B, C) \times hom_{\mathbb{C}}(A, B) \to hom_{\mathbb{C}}(A, C)$$

called *composition of morphisms*;

subject to the following axioms:

(i) the sets $hom_{\mathbb{C}}(A, B)$ are pairwise disjoint; that is, arrows uniquely determine their source and target objects;

(ii) composition is associative, that is, if $h \circ g$ and $g \circ f$ are both defined then

$$(h \circ g) \circ f = h \circ (g \circ f)$$

(iii) for each object A there is an identity morphism $Id_A \in hom_{\mathbb{C}}(A, A)$ which is a pre- and post-unit of composition:

$$Id_A \circ f = f \qquad g \circ Id_A = g$$

whenever the left sides are defined.

Definition B.2 (small category): A *small category* is a category in which the class of all morphisms in the category is a set.

Definition B.3 (initial object): An object $\mathbb{0}$ is an *initial object* in a category $\mathbb{C}$ if for every object A in $\mathbb{C}$, there is *exactly one* arrow from $\mathbb{0}$ to A. The unique arrows are written

$$!_A : \mathbb{0} \to A$$

The uniqueness ensures the following *universal property* of initial objects — for any $f : A \to B$,

$$\mathbb{0} \xrightarrow{!_A} A \xrightarrow{f} B = ¡_B$$

Terminal objects are the dual of initial objects:

Definition B.4 (terminal object): An object $\mathbb{1}$ is a *terminal object* in a category $\mathbb{C}$ if for every object A in $\mathbb{C}$, there is *exactly one* arrow from A to $\mathbb{1}$. The unique arrows are written

$$!_A : A \to \mathbb{1}$$

Terminal objects also have a universal property — for any $f : A \to B$,

$$A \xrightarrow{f} B \xrightarrow{!_B} \mathbb{1} = !_A$$

Definition B.5 (product): For two objects A and B in $\mathbb{C}$, a *product* is an object $A \times B$ together with two *projections*, $\grave{\pi} : A \times B \to A$ and $\acute{\pi} : A \times B \to B$ such that for any object $\mathbb{C}$ and pair of arrows $f : C \to A$ and $g : C \to B$ there is a unique arrow $f \vartriangle g : C \to A \times B$, called the *split* of f and g, that makes the following diagram commute:

$$\begin{tikzcd}
& C \arrow[dl, "f"'] \arrow[d, dotted, "f \vartriangle g"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\grave{\pi}"'] \arrow[r, "\acute{\pi}"] & B
\end{tikzcd}$$

Products have the following universal property — for any $h : C \to A \times B$,

$$h = f \vartriangle g \equiv (\grave{\pi} \circ h = f) \wedge (\acute{\pi} \circ h = g)$$

The dual of product is coproduct or sum:

Definition B.6 (coproduct, sum): For two objects A and B in $\mathbb{C}$, a *coproduct* or *sum* is an object $A + B$ together with two *injections*, $\grave{\iota} : A \to A + B$ and $\acute{\iota} : B \to A + B$ such that for any object $\mathbb{C}$ and pair of arrows $f : A \to C$ and $g : B \to C$ there is a unique arrow $f \triangledown g : A + B \to C$, called the *join* of f and g, that makes the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\grave{\iota}"] \arrow[dr, "f"] & A+B \arrow[d, dotted, "f \triangledown g"] & B \arrow[l, "\acute{\iota}"'] \arrow[dl, "g"'] \\
& C &
\end{tikzcd}$$

Sums have the dual universal property to products — for any $h : A + B \to C$,

$$h = f \triangledown g \equiv (h \circ \grave{\iota} = f) \wedge (h \circ \acute{\iota} = g)$$

Mappings between categories that preserve categorical structure are called *functors*:

Definition B.7 (functor): A *functor* $F : \mathbb{C} \to \mathbb{D}$ between two categories $\mathbb{C}$ and $\mathbb{D}$ consists of

(i) a map sending objects A in $\mathbb{C}$ to objects $F.A$ in $\mathbb{D}$;

(ii) a map sending arrows $f : A \to B$ in $\mathbb{C}$ to arrows $F.f : F.A \to F.B$ in $\mathbb{D}$.

The arrow mapping must preserve identities and composition:

(i) for every A in $\mathbb{C}$, $F.Id_A = Id_{F.A}$;

(ii) for all arrows f and g in $\mathbb{C}$, if $g \circ f$ is defined then $F.(g \circ f) = F.g \circ F.f$.

Definition B.8 (natural transformation): A *natural transformation* $\alpha : F \underset{\bullet}{\to} G$ between two functors $F, G : \mathbb{C} \to \mathbb{D}$ is a family of $\mathbb{D}$-arrows indexed by the objects of $\mathbb{C}$ such that for any arrow $f : A \to B$ in $\mathbb{C}$ the following *naturality square* commutes:

$$\begin{tikzcd}
F.A \arrow[r, "\alpha_A"] \arrow[d, "F.f"] & G.A \arrow[d, "G.f"] \\
F.B \arrow[r, "\alpha_B"] & G.B
\end{tikzcd}$$

For our purposes, a diagram is a functor from a small category:

Definition B.9 (diagram): If $\mathbb{J}$ is a small category then a functor $D : \mathbb{J} \to \mathbb{C}$ is a *diagram of shape* $\mathbb{J}$ *in* $\mathbb{C}$.

A *cone* α over a diagram $D : \mathbb{J} \to \mathbb{C}$ consists of an object A in $\mathbb{C}$ and for each object j in $\mathbb{J}$, an arrow $\alpha_j : A \to D.j$ such that for every arrow $f : j \to j'$ in $\mathbb{J}$,

$$\begin{tikzcd}
& D.j \arrow[dd, "D.f"] \\
A \arrow[ur, "\alpha_j"] \arrow[dr, "\alpha_{j'}"] & \\
& D.j'
\end{tikzcd}$$

A more succinct but equivalent definition is as a natural transformation from a constant functor to D:

Definition B.10 (cone): A *cone* over a diagram $D : \mathbb{J} \to \mathbb{C}$ is a natural transformation

$$\alpha : \underline{A} \underset{\bullet}{\longrightarrow} D$$

where $\underline{A}$ is the constant functor for some object A in $\mathbb{C}$.

We will often use the notation $\alpha : A \lhd D$ to mean $\alpha : \underline{A} \underset{\bullet}{\longrightarrow} D$ when we want to emphasize that we have a cone.

We have the dual notion of *cocone*:

Definition B.11 (cocone): A *cocone* under a diagram $D : \mathbb{J} \to \mathbb{C}$ is a natural transformation

$$\beta : D \underset{\bullet}{\longrightarrow} \underline{B}$$

for some object B in $\mathbb{C}$.

Our corresponding notation for a cocone $\beta : D \underset{\bullet}{\longrightarrow} \underline{B}$ is $\beta : D \rhd B$.

Definition B.12 (cone morphism): For cones over a diagram $D : \mathbb{J} \to \mathbb{C}$, a *cone morphism* between two cones $\alpha : A \lhd D$ and $\beta : B \lhd D$ is a $\mathbb{C}$-arrow $h : A \to B$ such that for each object j in $\mathbb{J}$,

$$\begin{tikzcd}
 & B \arrow[dd, "\beta_j"] \\
A \arrow[ur, "h"] \arrow[dr, "\alpha_j"] & \\
 & D.j
\end{tikzcd}$$

Cocone morphisms are defined dually. We can then define *categories of cones*:

Definition B.13 (cone category): For a diagram $D : \mathbb{J} \to \mathbb{C}$ define the *cone category* $Cone(D)$ with cones over D as objects and cone morphisms as arrows. Composition and identities are inherited from $\mathbb{C}$.

We can dually define *categories of cocones*.

Definition B.14 (limit or limiting cone): For a diagram $D : \mathbb{J} \to \mathbb{C}$, a *limit* or *limiting cone* over D is a terminal object in the category of cones $Cone(D)$.

If $\mu : U \lhd D$ is limiting cone over D, and $\alpha : A \lhd D$ is some other cone over D, then the unique cone morphism $h : U \to A$ from μ to α we will refer to as a *mediating morphism*.

Colimits are defined dually:

Definition B.15 (colimit or colimiting cocone): For a diagram $D : \mathbb{J} \to \mathbb{C}$, a *colimit* or *colimiting cocone* under D is an initial object in the category of cocones $Cocone(D)$.

References

[BackhouseJJM99]

Roland Backhouse, Patrik Jansson, Johan Jeuring and Lambert Meertens. Generic programming — an introduction. In S.D. Swierstra, P.R. Henriques and J.N. Oliveira (editors), *Proceedings of the 3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998.* Lecture Notes in Computer Science, volume 1608. Springer–Verlag, 1999.

[Backus78]

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21:8** (August 1978).

[Barr90]

Michael Barr and Charles Wells. *Category theory for computing science.* Prentice–Hall International, 1990.

[Bird87]

Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, International Summer School*, pages 5–42. Springer–Verlag, 1987.

[BirdM98]

Richard Bird and Lambert Meertens. Nested datatypes. In *Proceedings of the Mathematics of Program Construction 4th International Conference, Marstrand Sweden*, pages 52–67. Lecture Notes in Computer Science 1422, Johan Jeuring (editor). Springer–Verlag, June 1998.

[BirdP99]

Richard Bird and Ross Paterson. Generalized folds for nested datatypes. *Formal Aspects of Computing* **11** (2), 200–222 (1999).

[BirddM97]

Richard Bird and Oege de Moor. *Algebra of programming.* International Series in Computing Science, number 100. Prentice Hall, 1997.

[BosH88]

R. Bos and C. Hemerik. An introduction to the category-theoretic solution of recursive domain equations. Tech. Rep. Computing Science Notes 88 / 15 (October 1988), Eindhoven University of Technology, Department of Mathematics and Computing Science.

[BurstallD77]

R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery* **24**, 44–67 (1977).

[Bénabou85]

Jean Bénabou. Fibered categories and the foundations of naive category theory. *Journal of Symbolic Logic* **50** (1), 10–37 (1985).

[CockettD92]

Robin Cockett and Dwight Spencer. Strong categorical datatypes I. In *International Meeting on Category Theory 1991.* AMS, 1992.

[CockettF92]

Robin Cockett and Tom Fukushima. About Charity. Yellow Series. Tech. Rep. 92 / 480 / 18 (1992), Department of Computer Science, University of Calgary.

[FegarasS95]

Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions. Tech. Rep. 95/014 (June 1995), Oregon Graduate Institute.

[FiechH94]

Adrian Fiech and Michael Huth. Algebraic domains of natural transformations. *Theoretical Computer Science* **136** (1), 57–78 (December 1994).

[FiechS94]

Adrian Fiech and David A. Schmidt. Polymorphic lambda calculus and subtyping. Tech. Rep. TR–94–7 (1994), Kansas State University, Computing and Information Sciences Department.

[Fokkinga92]

M.M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1992.

[FokkingaM91]

Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Tech. Rep. 91-4 (1991), CWI, Amsterdam, Netherlands.

[Freyd90]

Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings of the 5th IEEE Annual Symposium on Logic in Computer Science, LICS'90, Philadelphia, PA, USA, 4–7 June 1990,* pages 498–507. IEEE Computer Society Press, 1990.

[GibbonsJ98]

Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the International Conference on Functional Programming, Baltimore, MD, September 1998*, 1998.

[Hagino87]

Tatsuya Hagino. *A categorical programming language*. Ph.D. thesis, University of Edinburgh, Department of Computer Science, September 1987.

[Hinze98]

Ralf Hinze. Numerical representations as higher-order nested data-types. Tech. Rep. IAI–TR–98–12 (December 1998), Institut für Informatik III, Universität Bonn.

[Hinze99]

Ralf Hinze. Manufacturing datatypes. Tech. Rep. IAI–TR–99–5 (April 1999), Institut für Informatik III, Universität Bonn.

[HoogendijkB97]

Paul Hoogendijk and Roland Backhouse. When do datatypes commute?. In E. Moggi and G. Rosolini (editors), *Proceedings of Category Theory and Computer Science*, pages 242–260. Lecture Notes in Computer Science, volume 1290. Springer–Verlag, 1997.

[Hutton98]

Graham Hutton. Fold and unfold for program semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September 1998*, 1998.

[Hutton99]

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* **9** (4), 355–372 (July 1999).

[Iverson62]

Kenneth E. Iverson. *A Programming Language*. John Wiley and Sons, 1962.

[Jacobs99]

B. Jacobs. *Categorical logic and type theory*. Studies in Logic and the Foundations of Mathematics, volume 141. Elsevier, 1999.

[JanssonJ97]

Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Conference record of the 24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL '97, Paris, France.* ACM Press, 1997.

[Jones98]

Mark P. Jones. Fixing algebraic datatypes. Slides from a talk given at University of Nottingham, 1998.

[JonesB99]

Mark P. Jones and P. Blampied. A pragmatic approach to maps and folds for parameterized datatypes. Unpublished draft, 1999.

[KubiakHL92]

Ryszard Kubiak, John Hughes and John Launchbury. Implementing projection-based strictness analysis. In *Proceedings of the Glasgow Functional Programming Workshop*. Springer–Verlag, 1992.

[LehmannS81]

Daniel J. Lehmann and Michael B. Smyth. Algebraic specification of data types: a synthetic approach. *Mathematical Systems Theory* **14**, 97–139 (1981).

[MacLane71]

Saunders Mac Lane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[MacLane88]

Saunders Mac Lane. To the greater health of mathematics. *The Mathematical Intelligencer* **10** (3), 17–20 (1988).

[Malcolm90]

Grant Malcolm. *Algebraic data types and program transformation.* Ph.D. thesis, Rijksuniversiteit Groningen, September 1990.

[ManesA86]

Ernest G. Manes and Michael A. Arbib. *Algebraic approaches to program semantics.* Texts and Monographs in Computer Science. Springer–Verlag, 1986.

[McCracken84]

Nancy McKracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen and G. Plotkin (editors), *Proceedings of the International Symposium on Semantics of Data Types, Sophia–Antipolis, France, June 1984.* Lecture Notes in Computer Science, number 173. Springer–Verlag, 1984.

[Meertens98]

Lambert Meertens. Functor pulling. In *Proceedings of the Workshop on Generic Programming, Marstrand, Sweden, 18th June 1998*, 1998.

[MeijerFP91]

Erik Meijer and Maarten Fokkinga and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, *Proceedings of the 5th ACM conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Lecture Notes in Computer Science 523. Springer, August 1991.

[MeijerH95]

Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Functional Programming Languages and Computer Architecture*. ACM, June 1995.

[Milner78]

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17** (3) (1978).

[MilnerTHM97]

Robin Milner, Mads Tofte, Robert Harper and David MacQueen. *The definition of Standard ML — revised*. MIT Press, 1997.

[Mycroft84]

Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet (editors), *International Symposium on Programming, 6th colloquium, Toulouse*, pages 217–228. Lecture Notes in Computer Science, volume 167, 1984.

[Okasaki98]

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[Okasaki99]

Chris Okasaki. From fast exponentiation to square matrices: an adventure in types. In *Proceedings of the International Conference on Functional Programming, 1999*, pages 28–35, 1999.

[PeytonJones]

Simon Peyton–Jones. Explicit quantification in Haskell. Unpublished notes. URL *http://research.microsoft.com/Users/simonpj/Haskell/quantification.html.*

[PeytonJonesH99]

Simon Peyton Jones and John Hughes (editors). Haskell 98: a non-strict, purely functional language. Tech. Rep. (February 1999). URL *http://www.haskell.org/definition/*.

[Phoa92]

Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Tech. Rep. (1992), Laboratory for the Foundations of Computer Science.

[Pierce91]

Benjamin C. Pierce. *Basic category theory for the computer scientist.* Foundations of Computing Series. The MIT Press, 1991.

[Plotkin83]

Gordon Plotkin. Domains. Electronic edition of the Pisa Notes on Domain Theory, 1983.

[Schmidt86]

D. A. Schmidt. *Denotational semantics: a methodology for language development.* Allyn and Bacon, Inc., 1986.

[Schubert72]

Horst Schubert. *Categories.* Springer–Verlag, 1972.

[SmythP82]

M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing* **11** (4), 761–783 (November 1982).

[TakanoM95]

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conference record of the 7th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, 25–28 June 1995*, pages 306–313. ACM Press, New York, 1995.

[TarleckiBG91]

Andrzej Tarlecki, Rod M. Burstall and Joseph A. Goguen. Some fundamental algebraic tools for the semantics of computation: part 3. Indexed categories. *Theoretical Computer Science* **91**, 239–264 (1991). Also, Monograph PRG–77, August 1989, Programming Resarch Group, Oxford University

[Taylor86]

Paul Taylor. *Recursive domains, indexed category theory and polymorphism.* Ph.D. thesis, Cambridge University, 1986.

[Tsuiki95]

Hideki Tsuiki. A denotational model of overloading. Tech. Rep. KSU / ICS–95–01 (1995), Department of Computer Science, Kyoto Sangyo University.

[Wadler89]

Philip Wadler. Theorems for free!. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. Addison-Wesley, Sept 1989.

[Wadler92]

Philip Wadler. The essence of functional programming. In *Principles of Programming Languages*, Jan 1992.

[Wand79]

M. Wand. Fixed point constructions in order-enriched categories. *Theoretical Computer Science* **8**, 13–30 (1979).