A Coalgebraic Approach to Lambda-calculus

Y. D. Duppen

E-mail address: `yduppen@xs4all.nl`
URL: `http://www.xs4all.nl/~yduppen`

Contents

Introduction

Wiskunde is een zeer respectabele wetenschap, hoewel men bij de beoefening ervan kan stuiten op perversiteiten, radicale idealen, duistere motieven en smerige integralen

(J. F. A. A. Nagel, RU Leiden)

Lambda-calculus is a theory often used in computer science. Its applications are manifold, ranging from proof-checkers to functional programming languages. Precisely because of these applications, infinitary lambda-calculus has recently become of interest among others at the Free University of Amsterdam.

In this thesis I will use a new approach to describing infinitary lambda-calculus; using coalgebraic techniques I hope to find a natural way to describe infinite terms and the operations on these.

The objectives in writing this thesis were not restricted to describing my results. Above all, I have tried to write something that is understandable to a somewhat broader public than just myself. In order to accomplish this, a large part of thesis consists of introductory material on the theories used for my description.

In the first chapter the concept of lambda-calculus will briefly be described. This chapter will merely explain the basics of the pure lambda calculus without so much as hinting at other lambda-calculi such as the typed lambda-calculus, the polymorphic lambda-calculus, and so on. Any trade-offs that had to be made between formal completeness and understandability were made in favor of the latter.

The next chapter deals with category theory, which forms the basis of coalgebra. Again, only those elements necessary for the final chapter have been described. Because category theory is very abstract, I have not tried to keep the definitions understandable; instead, examples have been provided for nearly every definition. This chapter can be read independently of the first.

The third chapter combines category theory with the notion of algebras and coalgebras. The basics of coalgebras will be explained and accompanied by the evergreen example of infinite lists.

Finally, in chapter four all these chapters will be combined into a coalgebraic account of lambda-calculus. Here, a coalgebra is created which maps to infinite trees in a natural way. The usual operations of substitution, one-step beta-reduction and its reflexive-transitive closure will be defined. As with other theories of infinite lambda-calculus, alpha-congruence will hardly be addressed. In the end the theory will be rewritten to De Bruijn terms, alleviating the problem of alpha-congruence. In order to keep the chapter readable some lengthy proofs can be found in appendix B.

Of course, a lot of people have helped me these past years during my studies. Of my teachers, I would like to thank Jan Willem Klop, Femke van Raamsdonk and Roel de Vrijer. They have first acquainted me with this subject and helped me all the way from understanding the basics to completing this thesis.

Furthermore, I would like to thank John Balder and Mark Lassche for proofreading the contents, and Désirée van den Berg, who proofread my English and saved the reader from my liberal views on interpunction.

Chapter 1

An Introduction to Lambda-calculus

Knights of the Lambda Calculus
A semi-mythical organization of wizardly LISP and Scheme hackers. The name refers to a mathematical formalism invented by Alonzo Church, with which LISP is intimately connected. There is no enrollment list and the criteria for induction are unclear, but one well-known LISPer has been known to give out buttons and, in general, the members know who they are....

(The Jargon File [**6**])

Lambda-calculus is a language of functions. Developed around 1930 by Alonzo Church it nowadays plays an important role in computer science. It is a powerful theory that (reputedly) can represent any kind of function, be it a high level function or not, typed or not, polymorphic or not ... The beauty of lambda-calculus lies in its simple mechanism of representing functions and 'executing' computations. Because of this simplicity it forms the basis of all functional programming languages.

This chapter will give an account of the so-called 'pure' lambda-calculus. It tries to cover enough ground to give an insight into lambda-calculus and to make clear what this thesis is about.

First the lambda-terms (functions) will be introduced by means of a structural inductive definition. Different notations of lambda-calculus are given, together with some examples. The text continues with describing the notion of alpha-congruence; this will make it possible to freely change certain variables.

After these basics, some time will be spent on substitution, i.e. replacement of certain variables by complete lambda-terms. Substitution is necessary to define the powerful notion of beta-reduction, the equivalence of 'computing'.

Finally, the De Bruijn notation will be dealt with. This way of describing lambda-terms is quite special in that it solves a lot of problems with substitution, is easy to understand for a computer, and is impossible to read for a human being.

All in all, this chapter will provide all the knowledge of lambda-calculus that is necessary to understand Chapter 4, in which another approach to lambda-calculus will be described.

1.1. Lambda-terms

Before we can do anything with lambda-calculus, it is necessary to define the elements of the calculus, the lambda-terms. They are inductively defined as follows:

Definition 1.1.1 (λ-terms)**.** The set Λ of λ-terms is the smallest set satisfying the following:

if x is a variable, then $x \in \Lambda$
if $M \in \Lambda$, then $(\lambda x.M) \in \Lambda$
if $M, N \in \Lambda$, then $(MN) \in \Lambda$

A term like $(\lambda x.M)$ is called an *abstraction over M.* It corresponds with a function that maps x to M. A term (MN) is called an *application.*

Example 1.1.2. Examples of λ-terms are:

$(\lambda y.z)$ $((\lambda x.(\lambda y.x))(\lambda z.z))$
(xy) $(\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$
$(x(\lambda x.((\lambda y.((zz)z))x)))$

As can be seen, the number of parentheses runs quickly out of hand. In order to avoid this, an alternative notation can be used. In this notation outer parentheses are elided and consecutive λ's are contracted; furthermore, abstraction associates to the right and application associates to the left.

Example 1.1.3. The alternative notation enables us to rewrite the λ-terms from Example 1.1.2 as follows:

$\lambda y.z$ $(\lambda xy.x)\lambda z.z$
xy $\lambda xyz.(xz)(yz)$
$x(\lambda x.(\lambda y.zzz)x)$

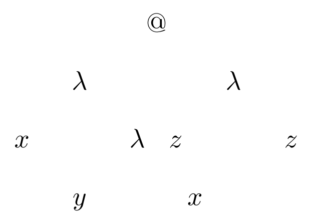

Figure 1.1.1. The tree-representation of $(\lambda xy.x)\lambda z.z$

Sometimes we draw λ-terms as trees. For instance the tree in Figure 1.1.1 represents the term $(\lambda xy.x)\lambda z.z$.

1.2. Variables and Alpha-congruence

As said before, abstractions correspond to functions. For example, the term $\lambda x.x$ corresponds to the function $f(x) = x$, the term $\lambda z.z$ corresponds to the function $f(z) = z$ and the term $\lambda x.y$ corresponds to the function $f(x) = y$. These examples are not arbitrary, as will be explained below.

If we look at the first two terms, we see that the functions they correspond to behave identically; they both map something to itself. However, syntactically they are different. The function of the last example is a strange one; it maps everything to y, whatever y might be. The abstracted variable x and the term y are not related at all.

In this section, first the relation between terms and their abstractions will be set out. Basically, if there is a term $\lambda x.M$ and x appears in M, it can be said that x is *bound in M*. If a variable is not bound in a term, it is said to be *free*. Formally, the notion of free and bound variables is defined as follows:

Definition 1.2.1. A variable x occurs free in a lambda-term M if it is not in the scope of a λx. It occurs bound otherwise.

Definition 1.2.2. The set $\mathsf{free}(M)$ which consists of the free variables in a lambda-term M is defined as:

$$\begin{aligned}\mathsf{free}(x) &= x\\ \mathsf{free}(\lambda x.M) &= \mathsf{free}(M) - \{x\}\\ \mathsf{free}(MN) &= \mathsf{free}(M) \cup \mathsf{free}(N)\end{aligned}$$

Example 1.2.3. When revisiting Example 1.1.2, the following holds:

Term M	Bound occurrences	Free occurrences	$\mathsf{free}(M)$
$\lambda y.z$	none	z	$\{z\}$
$(\lambda xy.x)\lambda z.z$	x and z	none	$\emptyset$
xy	none	x and y	$\{x, y\}$
$\lambda xyz.(xz)(yz)$	x, y and both zs	none	$\emptyset$
$x(\lambda xy.zzzx)$	all zs and the last x	the first x	$\{x\}$

As can be seen from this example, a variable may occur both bound and free in a lambda-term. We now introduce alpha-congruence.

Definition 1.2.4. M' is produced from M by a *change of bound variables* if a part $\lambda x.N$ of M is replaced by $\lambda y.N'$ where N' is obtained by replacing all occurrences of x in N by y. Here y is required to be fresh, i.e. it is not used in building N.

Definition 1.2.5 (α-congruence)**.** M is α-congruent with N, written as $M \equiv_\alpha N$, if N results from M by a series of changes of bound variables.

Example 1.2.6. According to the definition of α-congruence, we have that $x(\lambda xy.zzzx) \equiv_\alpha x(\lambda x_1 y.zzzx_1)$ and $\lambda x.x \equiv_\alpha \lambda z.z$. On the other hand we do not have $\lambda x.xy \equiv_\alpha \lambda y.yy$ because y already occurs in $\lambda x.xy$.

In the remainder of this chapter terms that are alpha-congruent are identified.

1.3. Substitution

This section will show how free variables in a λ-term can be replaced by other λ-terms. This process of replacing is called *substitution*. Substitution is necessary in order to define β-reduction which is the core of the pure lambda-calculus.

Substituting a term N for a variable x in M, written as $M[x := N]$ is not trivial, as the next example will show.

Example 1.3.1. Of the following substitutions:

$$(\lambda x.y)[y := \lambda z.zz] \equiv \lambda xz.zz$$
$$(\lambda xy.z)[z := yy] \equiv \lambda xy.yy$$
$$(\lambda xy.x)[x := z] \equiv \lambda xy.z$$

only the first substitution is well-behaved.

On the other hand, the second substitution is an example of *variable capture*. Here z is replaced by the term yy, in which both occurrences of y

are free. In the result of the substitution both occurrences of y are bound; they are said to have been 'captured' by the λy abstraction.

The third substitution is just plain wrong. On the left hand side, the occurrence of x is bound, while on the right hand side the occurrence of z is free.

In order to cope with the problem of variable capture we introduce the *Variable Convention*[1]. In effect, the Variable Convention says that variable capture can never occur.

Definition 1.3.2 (Variable Convention)**.** If $M_1, \ldots, M_n$ occur in a certain context then in these terms all bound variables are chosen to be different from free variables.

In the remainder of this chapter we will assume the Variable Convention.

Definition 1.3.3 (Substitution)**.** The result of substituting N for the free occurrences of x in M, notation $M[x := N]$ is defined as follows:

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y, \text{ if } x \not\equiv y \\ (\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]) \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_2[x := N]) \end{aligned}$$

In the third clause of this definition it is not necessary to add the condition 'provided that $x \not\equiv y$ and y does not occur in N'. This is already the case by the Variable Convention.

Example 1.3.4. It is e.g. possible to perform the following substitutions:

$$\begin{aligned} (\lambda yz.xzy)[x := (\lambda xz.zy)] &\equiv (\lambda y_1 z.xzy_1)[x := (\lambda xz_1.z_1 y)] \\ &\equiv \lambda y_1 z.(\lambda xz_1.z_1 y)zy_1 \end{aligned}$$

1.4. Beta-reduction

The notion of β-reduction lies at the heart of lambda-calculus. It is similar to computing the result of applying a function to an argument. Beta reduction is a relation on Λ.

[1]There is another solution. It can be found in section 1.6 below

Definition 1.4.1. A term P β-reduces to a term Q if $P \to_\beta Q$ can be derived from the following clauses:

(β) $$(\lambda x.M)N \to_\beta M[x := N]$$

(left) $$\frac{M \to_\beta M'}{NM \to_\beta NM'}$$

(right) $$\frac{M \to_\beta M'}{MN \to_\beta M'N}$$

(ξ) $$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

Example 1.4.2. We have e.g. the following reductions:

$$\begin{aligned}(\lambda xyz.xzy)\lambda xz.x &\equiv_\alpha (\lambda xyz.xzy)\lambda xw.x\\ &\to_\beta (\lambda yz.xzy)[x := \lambda xw.x]\\ &\equiv \lambda yz.(\lambda xw.x)zy\\ &\to_\beta \lambda yz.(\lambda w.z)y\\ &\to_\beta \lambda yz.z\end{aligned}$$

$$(\lambda xy.x)\lambda x.x \to_\beta \lambda yx.x$$

$$(\lambda x.xx)\lambda x.xx \to_\beta (\lambda x.xx)\lambda x.xx$$

We write β-reduction relation $\twoheadrightarrow_\beta$ for the reflexive-transitive closure of $\to_\beta$. While the one step β-reduction can only do one step[2], $\twoheadrightarrow_\beta$ can do zero or more steps at once. It can also be defined as follows:

Definition 1.4.3.

(ref) $$M \twoheadrightarrow_\beta M$$

$$\frac{M \to_\beta N}{M \twoheadrightarrow_\beta N}$$

(trans) $$\frac{M \twoheadrightarrow_\beta N \quad N \twoheadrightarrow_\beta L}{M \twoheadrightarrow_\beta L}$$

Example 1.4.4. We have for example:

$$(\lambda xyz.xzy)\lambda xz.x \twoheadrightarrow_\beta \lambda yz.z$$
$$(\lambda x.xx)\lambda x.xx \twoheadrightarrow_\beta (\lambda x.xx)\lambda x.xx$$

[2]This might be the reason it is called 'one step β-reduction'

but also

$$(\lambda xyz.xzy)\lambda xz.x \twoheadrightarrow_\beta (\lambda xyz.xzy)\lambda xz.x$$

Definition 1.4.5.

(1) A β-redex[3] is a term M, such that $M \equiv (\lambda x.N)N'$ for some N, N'.
(2) A term is called a β-normal form (nf), if it does not contain any redexes.
(3) A term M has a β-nf if there is a reduction $M \twoheadrightarrow_\beta N$ where N is a β-normal form.

Example 1.4.6. The term $\lambda x.x$ has no redexes; it is a normal form. The term $(\lambda xyz.xzy)\lambda xz.x$ has exactly one redex; it eventually reduces to the normal form $\lambda yz.z$. The term $\mathbf{\Omega} \equiv (\lambda x.xx)\lambda x.xx$ has one redex. However, it can only reduce to itself; $\mathbf{\Omega}$ has no normal form.

1.5. Church-Rosser

When computing a function, it is often desired that for a given input there is exactly one result. However, this is not necessarily so for the equivalent in lambda-calculus, β-reduction. When a term has multiple redexes, contracting different redexes might yield different results. This is not a problem *an sich*; after all, $2 * (2 + 4)$ also equals $2 * 6$ and $4 + 8$. It would be nice if these different λ-terms would eventually reduce to a unique term.

Definition 1.5.1 (Church-Rosser)**.** A reduction R is said to be Church-Rosser (CR), if for all lambda-terms M, N_1, N_2 such that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$ a term P exists such that $N_1 \twoheadrightarrow_\beta P$ and $N_2 \twoheadrightarrow_\beta P$.

With this definition in mind, the beginning of this section reduces to the question: is β reduction Church-Rosser? Luckily, the answer is yes. There is the following fundamental result, which is given without proof:

Theorem 1.5.2. *β-reduction is Church-Rosser.*

1.6. The De Bruijn Representation of Lambda-terms

One of the problems that were encountered with substitution was that of variable capture (see Section 1.3 above). The solution was to introduce the notion of α-congruence and assume a Variable Convention. There is however another solution which is invented by De Bruijn. Quite aptly, this is called the De Bruijn notation.

[3]This is an acronym for *red*ucible *ex*pression

De Bruijn noted that it is not necessary to give bound variables a name. All that is relevant for a variable is its connection with an abstraction. So instead of using variables inside λ-terms, De Bruijn used natural numbers. Each natural number corresponds with the distance to the corresponding λ. So for instance, in $\lambda xy.x$ all that is interesting about x is that it belongs to the first λ which is the second λ from x (counted from right to left). So he represented this term as $\lambda.\lambda.2$.

Remark. Technically, it could be said that the De Bruijn is just a convenient way of denoting an alpha-equivalence class of λ-terms.

Definition 1.6.1. Substitution on De Bruijn terms is defined as follows:

$$\begin{aligned}
n[m := N] &= \begin{cases} n & \text{if } n < m \\ n-1 & \text{if } n > m \\ \mathsf{rename}_{n,1}(N) & \text{if } n = m \end{cases} \\
(\lambda M)[m := N] &= \lambda(M[m+1 := N]) \\
(M_1 M_2)[m := N] &= (M_1[m := N])(M_2[m := N])
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{rename}_{m,i}(j) &= \begin{cases} j & \text{if } j < i \\ j+m-1 & \text{if } j \geq i \end{cases} \\
\mathsf{rename}_{m,i}(\lambda N) &= \lambda(\mathsf{rename}_{m,i+1}(N)) \\
\mathsf{rename}_{m,i}(N_1 N_2) &= (\mathsf{rename}_{m,i}(N_1)\ \mathsf{rename}_{m,i}(N_2))
\end{aligned}$$

And equation (β) on page 12 now becomes:

$$(\lambda P)Q = P[1 := Q] \qquad (\beta)$$

The resulting β-reduction behaves identical to that on normal λ-terms. For example:

Example 1.6.2. Just as $(\lambda xy.x)\lambda x.x \twoheadrightarrow_\beta \lambda yx.x$, we have that

$$\begin{aligned}
(\lambda.\lambda.2)\lambda.1 &\to_\beta (\lambda.2)[1 := \lambda.1] \\
&= \lambda.(2[2 := \lambda.1]) \\
&= \lambda.\,\mathsf{rename}_{2,1}(\lambda.1) \\
&= \lambda.\lambda.\,\mathsf{rename}_{2,2}(1) \\
&= \lambda.\lambda.1
\end{aligned}$$

While the De Bruijn notation does prevent nasty problems with substitution, the example above probably makes clear why in this paper it is avoided as much as possible; most humans find it hard to use[4].

[4]Computers have different opinions on this matter

1.7. References

This chapter is based on the book by C. Hankin [**4**]. It does cover a lot of ground and it has the advantage that it is very concise and descriptive at the same time. Many definitions and examples in this chapter come from this book. The others come from the book by Barendregt [**2**].

Proof of the fact that $\twoheadrightarrow_\beta$ is CR can be found in [**4**] too, where he uses a variant on the Tait-Martin-Löf proof. A shorter variant of this proof can be found in “Parallel Reductions in Lambda-calculus” [**19**].

Chapter 2

An Introduction to Category Theory

Man muss immer generalisieren

(Carl Jacobi [**14**])

Category theory is a relatively new branch in mathematics. Although it has only been in existence for about fifty years, it can be used to formalize notions that have been in use for hundreds of years; already it has a large number of applications, such as the design of functional programming languages, models of concurrency, type theory, polymorphism etc.

The theory itself is an abstract framework that could roughly be described as 'the science of thingies and arrows'. However, it turns out that it is a very powerful framework; it allows us to unite properties of different branches of mathematics into single notions that can easily be extended to other branches.

This advantage of category theory is also its largest drawback: the abstractness often makes it hard to understand what is going on. Fortunately, the basics of the theory are not too difficult and already give us a set of powerful tools.

This chapter will give a very short introduction to category theory. It starts with the definition of a category accompanied by some examples, formalizing the notion of thingies and arrows. It then continues with invertible arrows and isomorphisms. This subject is accompanied by an example of a categorical proof.

The next three sections deal with some categorical notions and their set-theoretic counterparts. The first is about initial and terminal objects. These objects play an important role in many categories, because of their special properties. The importance of terminal objects will be made clear in Chapter 3. After that, products and coproducts will be defined; these operators provide us with the means to combine different objects into another object. The last section gives the definition of a functor, a higher level function which can map categories to other categories.

2.1. The Category

The description of a category as “thingies and arrows” is mainly a cool slogan; its formal definition is given as follows:

Definition 2.1.1. A category $\mathcal{C}$ consists of:

(1) A collection of *objects*.

(2) A collection of *arrows*, also called *maps* or *morphisms*. An arrow f from object A to object B is denoted as $f : A \to B$ or $A \xrightarrow{f} B$. In this case, A is called the *domain* and B is called the *codomain*.

(3) A *composition* operator $\circ$ assigning to each pair of arrows $f : A \to B$ and $g : B \to C$ a *composite arrow* $g \circ f : A \to C$.

Composition is *associative*: for every set of arrows $f : A \to B$, $g : B \to C$ and $l : C \to D$ (with A, B, C and D not necessarily distinct), we have:

$$l \circ (g \circ f) = (l \circ g) \circ f$$

(4) For each object A an *identity* arrow $\mathsf{id}_A : A \to A$.

Identity satisfies the *identity law*: for any arrow $f : A \to B$, we have that:

$$\mathsf{id}_B \circ f = f \text{ and } f \circ \mathsf{id}_A = f$$

Example 2.1.2. As an example, let’s consider the category $\mathcal{O}$ of a small office, as depicted in Figure 2.1.1. It consists of four objects: E, the set of employees, C, the set of computers, H, the set of hostnames and I, the set of IP addresses. Furthermore, there are the arrows $c : E \to C$ which assigns a computer to to every employee, $i : C \to I$ which assigns an IP address to every computer and $h : I \to H$ which assigns a hostname to an IP address. Furthermore, for each object there is the identity map. This makes indeed a category because:

(1) it has a set of objects, $\{C, E, H, I\}$,

(2) it has a set of arrows, $\{c, h, i\}$,

(3) composition is associative; Figure 2.1.2 shows the result of $h \circ (i \circ c)$. It can easily be shown that this is equal to $(h \circ i) \circ c$,

Figure 2.1.1. The category $\mathcal{O}$

Figure 2.1.2. Picture of $h \circ (i \circ c)$

Figure 2.1.3. Picture of $\mathsf{id}_C \circ c$

(4) identity satisfies the identity law. For instance, Figure 2.1.3 shows that $\mathsf{id}_C \circ c = c$.

As the example shows, categorical proofs often consist of 'following the arrows'.

Example 2.1.3. A less contrived but well-known Example is $\mathcal{S}$, the category of sets where the objects are sets, the maps are the total functions from one set to another and each set has its own identity map.

And that is all there is. Category theory is completely based upon this sole definition. And just as Euclid's small set of axioms gives rise to an entire theory of geometry, so does this definition result in a vast array of abstract properties. Some of these properties will be discussed in the remainder of this chapter.

2.2. Inverses and Isomorphisms

Sometimes it is possible for a map to be inverted. For example, the map c from Example 2.1.2 assigns to each employee a computer, but there can equally well be a map $e : C \to E$ that assigns an employee to each computer. Just reverse the arrows of c.

Definition 2.2.1. An arrow $f : A \to B$ is an *isomorphism* if there is an arrow $g : B \to A$, called an *inverse* of f, such that:

$$g \circ f = \mathsf{id}_A \text{ and } f \circ g = \mathsf{id}_B$$

Two objects A and B are said to be *isomorphic*, written $A \cong B$, if there is an isomorphism between them.

The notion of isomorphisms is very important. Within categories it relates objects that behave identically.

It turns out that, if a map f has an inverse, this inverse is unique; it is denoted as f^{-1}. The proof of this is quite simple and it shows a lot about the beauty of many categorical proofs.

Proposition 2.2.2. *If $g : B \to A$ and $k : B \to A$ are both inverses for $f : A \to B$, then $g = k$.*

Proof. If g and k are both inverses, then we have that:

$$\begin{aligned} g \circ f &= \mathsf{id}_A & \qquad k \circ f &= \mathsf{id}_A \\ f \circ g &= \mathsf{id}_B & \qquad f \circ k &= \mathsf{id}_B \end{aligned}$$

by the definition of inverse. Then it follows that:

$$\begin{aligned} k &= \mathsf{id}_A \circ k && \text{by the identity law} \\ &= (g \circ f) \circ k && \text{because } g \text{ is an inverse} \\ &= g \circ (f \circ k) && \text{because composition is associative} \\ &= g \circ \mathsf{id}_B && \text{because } k \text{ is an inverse} \\ &= g && \text{by the identity law} \end{aligned}$$

so indeed $k = g$. □

2.3. Initial and Terminal Objects

This section will describe two other universal constructions in categories, initial and terminal objects. Special properties of these objects form the basis of the rest of this paper.

Definition 2.3.1. An object A of a category $\mathcal{C}$ is said to be an *initial object* of $\mathcal{C}$ if for each object X of $\mathcal{C}$ there is exactly one map $A \to X$.

Definition 2.3.2. An object Z of a category $\mathcal{C}$ is said to be a *terminal object* or *final object*[1] if for each object X of $\mathcal{C}$ there is exactly one map $X \to Z$.

These definitions look similar and indeed they are. The notion of initial objects is said to be the *dual* of final objects. Duality plays an important role in relating algebras and coalgebras in the next chapter.

Initial and final objects both have interesting properties, as the following two propositions will show:

Proposition 2.3.3. *If Z_1 and Z_2 are both terminal objects in category $\mathcal{C}$, then there is exactly one map $Z_1 \to Z_2$, and that map is an isomorphism.*

Proof. By the definition of finality, there is exactly one map $Z_1 \to Z_2$. But by the same definition, there is also exactly one map $Z_2 \to Z_1$, because Z_1 is final too.

Composing these maps, results in the following maps:

$$Z_1 \to Z_2 \to Z_1$$
$$Z_2 \to Z_1 \to Z_2$$

These maps are unique because Z_1 and Z_2 are final. According to the definition of a category there also exists a unique map $\mathsf{id}_{Z_1} : Z_1 \to Z_1$. This means that

$$Z_1 \to Z_2 \to Z_1 = \mathsf{id}_{Z_1}$$

and also

$$Z_2 \to Z_1 \to Z_2 = \mathsf{id}_{Z_2} \tag{2.3.1}$$

This means that the maps $Z_1 \to Z_2$ and $Z_2 \to Z_1$ are inverses of each other. And so, by definition of isomorphisms, $Z_1 \to Z_2$ is an isomorphism. □

Proposition 2.3.4. *If A_1 and A_2 are both initial objects in category $\mathcal{C}$, then there is exactly one map $A_1 \to A_2$, and that map is an isomorphism.*

Proof. This proof is similar to the proof of Proposition 2.3.3. □

[1] I agree with [**17**] that *final* sounds a lot better than *terminal*

$$\begin{tikzcd}
& & X & & \\
& f_1 & f = \langle f_1, f_2 \rangle & f_2 & \\
B_1 & \pi_1 & B_1 \times B_2 & \pi_2 & B_2
\end{tikzcd}$$

Figure 2.4.1. Definition of a product $P = B_1 \times B_2$

2.4. Combining Objects: Products and Coproducts

The possibility to combine different objects into one is a powerful concept. Many people do it on a daily basis, often without even realising it. For example, the index of an atlas consists of entries like: ‘Amsterdam, 14B8’, where the city of Amsterdam is combined with page 14, row B and column 8. Another example is the concept of fruit. A piece of fruit is either an apple, or a banana, or . . . , but never more than one.

These concepts of combining objects are also present in category theory, albeit in an abstract way. However, when these concepts are translated back to $\mathcal{S}$, the category of sets, they behave exactly as described above.

We begin with the notion of product, of which the atlas was an example:

Definition 2.4.1. An object P together with pair of maps $\pi_1 : P \to B_1$ and $\pi_2 : P \to B_2$ is called a *product* of B_1 and B_2 if for each object X and each pair of maps $f_1 : X \to B_1$, $f_2 : X \to B_2$ there is exactly one map $f : X \to P$ such that $f_1 = \pi_1 \circ f$ and $f_2 = \pi_2 \circ f$ (Figure 2.4.1).

Lemma 2.4.2. *If $(P, \pi_1\pi_2)$ and (P', π_1', π_2') are both products of A and B, then $P \cong P'$.*

Proof. Since P and Q are both products of A and B, we can draw the solid lines in Figures 2.4.2 (where P has been drawn twice). By definition for each object X and each pair of maps $f_1 : X \to A$ and $f_2 : X \to B$ there is exactly one map $f : X \to P$. In other words, there is a unique map $\langle \pi_1', \pi_2' \rangle : Q \to P$. By the same line of reasoning, there is also a unique map $\langle \pi_1, \pi_2 \rangle : P \to Q$.

This means that the map $\langle \pi_1', \pi_2' \rangle \circ \langle \pi_1, \pi_2 \rangle : P \to P$ is also unique and therefore equal to $\mathsf{id}_P : P \to P$. In other words, $\langle \pi_1', \pi_2' \rangle$ and $\langle \pi_1, \pi_2 \rangle$ must be inverses of each other. Which means that $P \cong Q$. $\square$

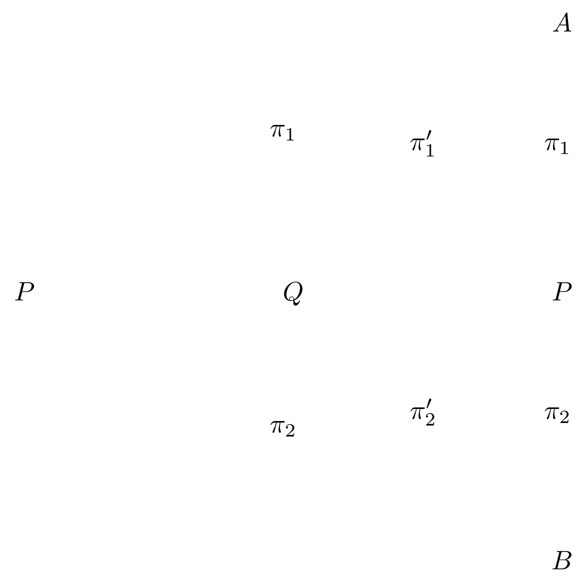

Figure 2.4.2. Proving Proposition 2.4.2

The previous lemma allows us to speak of *the* product of A and B. This product is often written as $A \times B$, as in Figure 2.4.1. The unique map f that was referred to in Definition 2.4.1 is written as $\langle f_1, f_2 \rangle$.

The other construct introduced here is the sum of two objects, of which the fruit was an example. It is interesting to note that its definition is similar to that of the product. Indeed, the sum is the dual of the product. For that reason, sums are often called *coproducts*.

Definition 2.4.3. An object S together with a pair of maps $\kappa_1 : B_1 \to S$ and $\kappa_2 : B_2 \to S$ is called a *sum* of B_1 and B_2 if for each object Y and each pair of maps $g_1 : B_1 \to Y$, $g_2 : B_2 \to Y$ there is exactly one map $g : S \to Y$ such that $g_1 = g \circ \kappa_1$ and $g_2 = g \circ \kappa_2$ (Figure 2.4.3).

Lemma 2.4.4. *If (S, κ_1, κ_2) and $(S', \kappa_1', \kappa_2')$ are both sums of A and B, then $S \cong S'$.*

Proof. Similar to the proof of Lemma 2.4.2. □

Just as Lemma 2.4.2 allowed us to speak of *the* product of A and B, so does this lemma allow us to speak of *the* sum of A and B, written as $A + B$. The unique map g that the definition refers to is written as $[g_1, g_2]$.

$$\begin{tikzcd}
B_1 \arrow[r, "\kappa_1"] \arrow[dr, "g_1"'] & B_1 + B_2 \arrow[d, "g = [g_1, g_2]"] & B_2 \arrow[l, "\kappa_2"'] \arrow[dl, "g_2"] \\
 & Y &
\end{tikzcd}$$

Figure 2.4.3. Definition of a sum $S = B_1 + B_2$

Lemma 2.4.5. *For sums and products, the following equalities hold (among others):*

$$A \times B \cong B \times A \qquad A + B \cong B + A$$
$$A \times (B \times C) \cong (A \times B) \times C \qquad A + (B + C) \cong (A + B) + C$$

And in some categories, called distributive *categories, such as the category of sets $\mathcal{S}$, the following holds too:*

$$A \times (B + C) \cong (A \times B) + (A \times C)$$

This lemma allows us to leave out a lot of brackets; this means that we can write e. g. $A \times B \times (A + B + C) \times D$.

In $\mathcal{S}$, the category of sets, a product of two sets is equal to the classical *Cartesian product*, while the sum of two sets is equal to the *disjoint sum*.

2.5. Functors

Everything can be turned into a category, as long as it satisfies the requirements in Definition 2.1.1. It turns out that it is also possible to create a category of categories $\mathcal{C}^{\mathcal{C}}$. In this category, the objects are categories and the map are *functors*.

Definition 2.5.1. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A *functor* $F : \mathcal{C} \to \mathcal{D}$ is a map taking each $\mathcal{C}$-object A to a $\mathcal{D}$-object $F(A)$ and each $\mathcal{C}$-arrow $A \xrightarrow{f} B$ to a $\mathcal{D}$-arrow $F(A) \xrightarrow{F(f)} F(B)$, such that for all $\mathcal{C}$-objects A and composable $\mathcal{C}$-arrows f and g

(1) $F(\mathsf{id}_A) = \mathsf{id}_{F(A)}$
(2) $F(g \circ f) = F(g) \circ F(f)$.

Example 2.5.2. Let F be the functor defined by

$$F(X) = 1 + (A \times X)$$

for a given set A and an arbitrary singleton set 1. Let us further assume that we have a map

$$m : A^\infty \times A^\infty \to A^\infty$$

which maps the product of two infinite lists (elements of A^∞) to another infinite list.

Applying F to this map results in the map

$$F(m) : 1 + (A \times A^\infty \times A^\infty) \to 1 + (A \times A^\infty)$$

with $F(m) = [\mathsf{id}_1, \langle \mathsf{id}_A, m \rangle]$.

Remark. It has been explained that functors map objects to other objects. This means that their domain and codomain are categories. In this thesis, functors will always work on the category of sets. From now on, every functor F will implicitly be defined as

$$F : \mathcal{S} \to \updownarrow \mathcal{S}$$

2.6. References

Category theory requires a different way of thinking than classical set theory. I found the book “Conceptual Mathematics, a first Introduction to Mathematics” by Lawvere and Schanuel [**7**] a more than excellent book in this regard. They have a relaxed attitude towards the subject and do not try to cram as much information in as few pages as possible. The book is full of pictures, examples and exercises. In my opinion, it is by far the best book I have read on the subject of mathematics.

The book does have one drawback: it is not very suitable as a reference book. Luckily, “Lambda Calculi, a Guide for Computer Scientists” by Benjamin Pierce [**15**] can be used for that purpose. It’s style is similar to that of Hankin described in the previous chapter, in that it is very concise. Most definitions in this chapter are based on Pierce’s book.

Both books describe many more concepts than those described in this chapter. Proofs of the lemmas in this chapter can be found there too. Funnily enough, Pierce describes more concepts than Lawvere and Schanuel, even though it contains only one-third as many pages.

Chapter 3

An Introduction to Coalgebras and Coinduction

The mathematical sciences particularly exhibit order, symmetry, and limitation; and these are the greatest forms of the beautiful.

(Aristotle [**14**])

The formalisms in the previous chapter are also suitable to represent *algebras*. Algebras are mathematical constructs where you can combine small building blocks into larger blocks; these blocks can then be used to build even larger blocks, which can be used to build still larger blocks and so on. However, this 'and so on' can not go on forever, making it difficult to construct blocks that are infinitely large.

That is were coalgebras come into play. Usually, algebras are described as in Chapter 1. However, it is also possible to describe them categorically. Coalgebras can then be seen as the dual of algebras. They are as powerful as algebras, and it turns out that they have the advantage of being able to deal with infinite terms. Coalgebras are the subject of this chapter.

The first section describes how an algebra can be described using categorical notions. This section does not contain formal definitions and merely serves to show that algebras and coalgebras are related. The next section gives a formal definition of a coalgebra. While not all coalgebras are useful, some coalgebras are *final*, and it turns out that this property is very important. The definitions are accompanied by an example of infinite streams.

The remainder of the chapter deals with *coinduction*, the dual of induction. Coinduction can be used to define new functions and to prove equalities. The latter is done by introducing *bisimulations*. These notions too are clarified using the example of infinite streams.

The chapter finishes with the Flattening Lemma, a property that will turn out to be useful in the next chapter. The proof of the lemma will be given too; it is a beautiful example of the category theory involved in coalgebras.

3.1. Lambda Calculus revisited

The definition of λ-terms in Section 1.1 is an example of an *inductive definition*. It gives the smallest possible terms (variables) and it gives two rules to combine existing terms into larger ones (abstraction and application).

But instead of looking at this definition as a set of rules, it can also be seen as a set of three *maps*: the map $\mathsf{lambda}_{\mathsf{var}} : V \to \Lambda$, which turns a variable into a λ-term, the map $\mathsf{lambda}_{\mathsf{abs}} : V \times \Lambda \to \Lambda$ which combines a variable and an existing λ-term into a new λ-term, and the map $\mathsf{lambda}_{\mathsf{app}} : \Lambda \times \Lambda \to \Lambda$ which combines two existing λ-terms into a new λ-term, all with the expected definitions.

Since these three maps all have the same codomain, they can be joined together into the map

$$\mathsf{lambda} : V + (V \times \Lambda) + (\Lambda \times \Lambda) \to \Lambda$$

where $\mathsf{lambda} = [\mathsf{lambda}_{\mathsf{var}}, \mathsf{lambda}_{\mathsf{abs}}, \mathsf{lambda}_{\mathsf{app}}]$. The function lambda is a *constructor*: it can be used to *construct* new terms out of existing terms.

Another point worth of noting is that the domain and the codomain share a common factor, Λ. So if we introduce the functor F, defined as

$$F(X) = V + (V \times X) + (X \times X)$$

lambda can be written as

$$\mathsf{lambda} : F(\Lambda) \to \Lambda$$

If so desired, it is now possible to create the *category of F-algebras*, where the objects are the algebras $a : F(U) \to U$ and the mappings are homomorphisms between algebras. The next sections will describe such a category, but it will be the category of *F-coalgebras*.

3.2. Coalgebras

While algebras can be seen as methods to construct structures, coalgebras are used to *destruct* them. Essentially, they are the duals of algebras. It is the difference between a mason and an archaeologist. A mason has a set of stones, from which he can create the most marvellous buildings. An archaeologist, on the other hand, encounters existing buildings and takes them apart, very carefully so as not to destroy any information[1]. But while their approaches differ, at the end they both know the entire building. Coalgebras are the archaeologists of mathematical structures.

Definition 3.2.1. Let F be a functor. An F-coalgebra (or coalgebra for short) is a pair (U, c) consisting of a set U and a function $c : U \to F(U)$.

Here U is called the *carrier* and the function c is called the *structure* or *operation* of the F-coalgebra.

Example 3.2.2. Let F be the functor

$$F(X) = A \times X$$

for a given set A. For the set A^∞ of infinite lists, which are made up of elements of A, it is possible to create the F-coalgebra $(A^\infty, \langle \mathsf{head}, \mathsf{tail} \rangle)$.

This coalgebra is made by combining the functions $\mathsf{head} : A^\infty \to A$ and $\mathsf{tail} : A^\infty \to A^\infty$ into the function $\langle \mathsf{head}, \mathsf{tail} \rangle : A^\infty \to A \times A^\infty$.

The intuition is that the function head maps an infinite list of elements of A to its first element, which is in this way observed, and that the function tail maps the infinite list to its tail, i.e. the original list without its first element. Since the tail is an infinite list too, we can apply head and tail again and again.

3.3. The Category of Coalgebras

Coalgebras are not much fun in themselves. However, when we create the category $\mathcal{F}$, the category of F-coalgebras, the real fun begins. In order to define $\mathcal{F}$, it is necessary to define its objects and its maps. But first homomorphisms of coalgebras are introduced.

Definition 3.3.1. A *homomorphism of coalgebras* (or *map of coalgebras*, or *coalgebra map*) from an F-coalgebra (U_1, c_1) to another F-coalgebra (U_2, c_2) is a function $h : U_1 \to U_2$ between the carriers that has the property that:

$$c_2 \circ h = F(h) \circ c_1$$

as expressed in Figure 3.3.1.

It turns out that homomorphisms of coalgebras have exactly the desired properties; they can be composed and they are associative.

Definition 3.3.2. The category $\mathcal{F}$ is the category with F-coalgebras as its objects and *homomorphisms of coalgebras* as its arrows.

[1]Of course archaeologists do not do this all the time. But sometimes they have to, and they make a nice metaphor

$$\begin{tikzcd}
U_1 \arrow[r, "h"] \arrow[d, "c_1"'] & U_2 \arrow[d, "c_2"] \\
F(U_1) \arrow[r, "F(h)"'] & F(U_2)
\end{tikzcd}$$

Figure 3.3.1. A homomorphism of coalgebras

$$\begin{tikzcd}
U \arrow[r, "h"] \arrow[d, "\langle \mathsf{value}, \mathsf{next} \rangle"'] & A^{\infty} \arrow[d, "\langle \mathsf{head}, \mathsf{tail} \rangle"] \\
A \times U \arrow[r, "\mathsf{id}_A \times h"'] & A \times A^{\infty}
\end{tikzcd}$$

Figure 3.3.2. The homomorphism h

Remark. Those who have read "Conceptual Mathematics" [**7**] will notice that $\mathcal{F}$ is very similar to $\mathcal{S}^{\downarrow\downarrow}$, the category of irreflexive graphs.

Definition 3.3.3. A *final F-coalgebra* (W, d) is a coalgebra such that for every F-coalgebra (U, c) there is a unique homomorphism of coalgebras $(U, c) \to (W, d)$.

The last definition might look very similar to Definition 2.3.2. This is no coincidence. Final coalgebras are the terminal objects in $\mathcal{F}$.

Example 3.3.4. The previous example is continued by showing that the coalgebra $(A^{\infty}, \langle \mathsf{head}, \mathsf{tail} \rangle)$ is a final coalgebra for the functor

$$F(X) = A \times X$$

Let $\langle \mathsf{value}, \mathsf{next} \rangle : U \to F(U)$ be an arbitrary F-coalgebra. Firstly, we show that there exists a homomorphism from U to A^{∞}. It is given by

$$h(u)(n) = \mathsf{value}(\mathsf{next}^{(n)}(u))$$

Obviously, h is a function. However, since the following holds too:

$$\langle \mathsf{head}, \mathsf{tail} \rangle \circ h = (\mathsf{id}_A \times h) \circ \langle \mathsf{value}, \mathsf{next} \rangle$$

as in Figure 3.3.2, it is also a homomorphism. It can also be proven[2] that this homomorphism is unique by assuming that there is also a function $h' : U \to A^{\infty}$ and showing that this function is identical to h.

[2] *[Huius] rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.* — I have discovered a truly marvelous demonstration of this proposition (but) this margin is too narrow to contain it.

$$\begin{tikzcd} U \arrow[r, "\exists! f"] \arrow[d, "\forall c"'] & W \arrow[d, "d", "\cong"'] \\ F(U) \arrow[r, "F(f)"'] & W(U) \end{tikzcd}$$

Figure 3.3.3. Definition by finality of f

$$\begin{tikzcd} A^\infty \arrow[r, "\mathsf{odd}"] \arrow[d, "\lambda\alpha.(\mathsf{head}(\alpha),\mathsf{tail}(\mathsf{tail}(\alpha)))"'] & A^\infty \arrow[d, "\langle \mathsf{head},\mathsf{tail}\rangle", "\cong"'] \\ A \times A^\infty \arrow[r, "\mathsf{id}_A \times \mathsf{odd}"'] & A \times A^\infty \end{tikzcd}$$

Figure 3.3.4. Coinductive definition of the function odd

Final coalgebras have two important properties. The first property is not confined to final coalgebras; it is a property of all final objects as can be seen in Proposition 2.3.3.

Lemma 3.3.5. *Final coalgebras, if they exist, are uniquely determined (up-to-isomorphism).*

Lemma 3.3.6. *A final coalgebra $W \to F(W)$ is the greatest fixed point $W \xrightarrow{\cong} F(W)$ of the functor F.*

The last lemma basically says that, if W is a carrier of a coalgebra, it is the largest possible set in which each element can be decomposed into other elements.

Finality of a coalgebra can be used to define functions into the carrier. Let us assume that (W, d) is a final F-coalgebra, and that we wish to create a function $f : U \to W$. If an F-coalgebra (U, c) can be created, there is exactly one function from U to W (Figure 3.3.3).

Example 3.3.7. This example continues Example 3.2.2 on infinite lists. Suppose we want to define a function odd that takes an infinite list and produces a new infinite list. This new list contains all the elements occurring in oddly numbered places of the original list. This function should have the following observations:

$$\mathsf{head}(\mathsf{odd}(\alpha)) = \mathsf{head}(\alpha) \quad \text{and} \quad \mathsf{tail}(\mathsf{odd}(\alpha)) = \mathsf{odd}(\mathsf{tail}(\mathsf{tail}(\alpha)))$$

It can be defined coinductively, that is, by finality of A^∞ as in Figure 3.3.4:

$$\begin{tikzcd}
U \arrow[d, "c"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\gamma"] & U \arrow[d, "c"] \\
T(U) & T(R) \arrow[l, "T(\pi_1)"] \arrow[r, "T(\pi_2)"'] & T(U)
\end{tikzcd}$$

Figure 3.4.1. The bisimulation R

The unique coalgebra homomorphism that arises here can be called odd. It can easily be shown that it has the desired observations.

3.4. Bisimulations

Proving that two infinite structures are equal can be done using bisimulations. Bisimulations are relations on structures that use the same technique as coalgebras: observing and going on.

Example 3.4.1. Bisimulations can be used to prove that two infinite lists are equal. A bisimulation on the carrier A^∞ is a relation R on A^∞ satisfying

$$R(\alpha, \beta) \implies \begin{cases} \mathsf{head}(\alpha) = \mathsf{head}(\beta) \text{ and} \\ R(\mathsf{tail}(\alpha), \mathsf{tail}(\beta)) \end{cases}$$

In other words, two lists are bisimilar if their heads are equal and their tails are bisimilar too.

While the bisimulation in the previous example is quite *ad hoc*, there is a more formal definition.

Definition 3.4.2. Let F be a functor, and (U, c) an F-coalgebra. A bisimulation on U is a relation R on U for which there exists an F-coalgebra structure $\gamma : R \to T(R)$ such that the two projection functions π_1 and π_2 are homomorphisms of F-coalgebras (Figure 3.4.1).

The fact that bisimulation implies equality is captured in the *coinduction proof principle*:

Theorem 3.4.3. *Let $c : U \to F(U)$ be the final F-coalgebra. For all v and w in U, if $R(v, w)$, for some bisimulation R on U, then $v = w$.*

There can be many bisimulations on a final F-coalgebra. However, when one is not interested in a specific bisimulation, it is always possible to talk of the greatest bisimulation.

Proposition 3.4.4. *The greatest bisimulation on a carrier U always exists, and it is denoted by $\sim_U$ or $\sim$ for short. It is the union of all bisimulations.*

$$\begin{tikzcd}
U \arrow[r, "f"] \arrow[d, "c"'] & W \arrow[d, "\cong"', "d"] \\
F(U) \arrow[r, "F(f)"'] & F(W)
\end{tikzcd}$$

Figure 3.5.1. Defining a function $f : U \to W$

$$\begin{tikzcd}
W \arrow[r, "\exists ! f"] \arrow[d, "\forall f'"'] & U \arrow[d, "\cong"', "c"] \\
F(V+W) \arrow[r, "F({[g^*, f]})"'] & F(U)
\end{tikzcd}$$

Figure 3.5.2. The Flattening Lemma

3.5. The Flattening Lemma

The concept of coinductive definitions can be very powerful, as can be seen in the next chapter, but it has some drawbacks. To define a function $f : U \to W$ to an F-coalgebra (W, d), a coalgebra (U, c) must be created which also maps to $F(U)$ (Figure 3.5.1). However, sometimes the codomain of c is more complex.

This problem can be overcome by the Flattening Lemma. This lemma states that instead of using a function $c : U \to F(U)$ it is also possible to use a function $c : U \to F(V + U)$. This will turn out to be very important when substitution is defined in the next chapter.

The proof of the Flattening Lemma is very elegant. It shows the use of describing the category $\mathcal{F}$ by merely using categoric techniques in order to obtain a coalgebraic result.

Lemma 3.5.1 (The Flattening Lemma)**.** *Let $c : U \to F(U)$ be a final coalgebra, let $g : V \to F(V)$ be an arbitrary F-coalgebra and let $g^* : V \to U$ be the unique homomorphism between the two. Consider any $f' : W \to F(V + W)$. Then there is a unique map $f : W \to U$ such that*

$$c \circ f = F([g^*, f]) \circ f' \tag{3.5.1}$$

This is depicted in Figure 3.5.2.

Figure 3.5.3. Proving the Flattening Lemma

Proof. We "flatten" f to $[F(\kappa_1) \circ g, f'] : V + W \to F(V + W)$. This is an F-coalgebra, so there is a unique homomorphism $h : V + W \to U$. Now

$$\begin{aligned} c \circ (h \circ \kappa_1) &= F(h) \circ ([F(\kappa_1) \circ g, f'] \circ \kappa_1) \\ &= F(h) \circ (F(\kappa_1) \circ g) \\ &= F(h \circ \kappa_1) \circ g \end{aligned}$$

So by finality, $h \circ \kappa_1 = g^*$. Let $f = h \circ \kappa_2$. Then

$$\begin{aligned} c \circ f &= c \circ (h \circ \kappa_2) \\ &= F(h) \circ ([F(\kappa_1) \circ g, f'] \circ \kappa_2) \\ &= F(h) \circ f' \\ &= F([g^*, f]) \circ f' \end{aligned}$$

It follows that f has the desired properties (Figure 3.5.3).

In order to prove that f is unique, suppose that f'' satisfies equation (3.5.1). Then $[g^*, f'']$ would be a coalgebra morphism from $V + W$ to U. By finality, $[g^*, f''] = h = [g^*, f]$. So $f'' = f$. □

3.6. References

My first introduction to category theory was "A Tutorial on (Co)Algebras and (Co)Induction" by Bart Jacobs and Jan Rutten [**8**]. It is a good article with a structure similar to this chapter; they begin by categorically describing algebras, after which they move on to coalgebras. The article is full of examples, among others the infinite list example used above.

The article "Universal Coalgebras" [**17**] is less accessible, but it covers more ground. "Exercises in Coalgebraic Specification" [**5**] is more of a case study and it shows the power of coalgebras in action.

After much struggling with substitution in the next chapter, I encountered the Flattening Lemma in Lawrence Moss' article "Parametric Corecursion" [**13**]. It is a general article on corecursion, that builds upon the Flattening Lemma.

Chapter 4

A Coalgebraic Approach to Lambda-calculus

Only two things are infinite, the universe and human stupidity, and I'm not sure about the former.

(Albert Einstein [**18**])

In this chapter, the previous three chapters will be combined to give a coalgebraic account of possibly infinite lambda-terms, such as $\lambda x.xxx\ldots$, $\mathsf{I}(\mathsf{I}(\mathsf{I}(\mathsf{I}\ldots)))$ etc. Infinite lambda-terms often arise in the semantics of lazy funtional languages. For example, in such a language it is perfectly acceptable to filter all primes from the (infinite) list of all natural numbers.

It is not new to describe an infinitary lambda-calculus; "Infinitary Lambda Calculus" [**9**] is an article on this subject in which the authors model infinite trees within an (ultra-) metric space. By using this approach they identify several kinds of metrics that correspond with several kinds of infinitary lambda-calculus, each with its own properties.

M. Lenisa has another approach in her article "Semantic Techniques for Deriving Coinductive Characterizations of Observational Equivalences for λ-calculi" [**11**]; she compares different reduction strategies on possibly infinite terms. This approach assumes the existence of infinite terms and reductions on these terms. Furthermore, it only compares different types of reduction.

Lambek and Scott give a categoric account of combinatory logic in “Introduction to Higher Order Categorical Logic”[**10**]. This book is an introduction to categorical logic, and their combinators merely serve as an example. They have the advantage that combinators are closed lambda-terms, which makes reduction a lot easier.

This chapter will describe yet another approach. Lambda-terms are viewed as possibly infinite trees which can be represented by an appropriate functor and structure. It will be shown that the set $\overline{\Lambda_T}$ of possibly infinite lambda-trees together with this structure is a final coalgebra.

Based on these results a coinductive definition of substitution will be given. The problems associated with free variables will be ignored and the notion of alpha-congruence will be assumed. It turns out that the Flattening Lemma makes the definition quite elegant. The text continues with defining beta-reduction.

Finally, a similar definition of possibly infinite De Bruijn trees will be given. As it turns out, many results from this chapter also apply to De Bruijn trees, albeit with some little modifications.

4.1. Describing Lambda-terms

As described in Chapter 1, it is easy to represent an arbitrary λ-term as a tree. To recapitulate, λ-trees are trees in which each node consists of either:

(1) a leaf node (a variable: x),

(2) a leaf node and another subtree (an abstraction: $\overset{\lambda}{\underset{x}{\diagup}\ \underset{M}{\diagdown}}$), or

(3) two subtrees (an application: $\overset{@}{\underset{M}{\diagup}\ \underset{N}{\diagdown}}$)

This tree-structure can be captured in a functor T:

Definition 4.1.1. The λ-tree structure can be represented with the functor T, defined as

$$T(X) = V + (V \times X) + (X \times X) \tag{4.1.1}$$

In order to create a coalgebra from this functor we need to find a suitable structure and a carrier. As the carrier we'd like to use the set $\overline{\Lambda_T}$, the set of all finite and infinite lambda-trees.

To complete our definition of the coalgebra we define the map unfold : $\overline{\Lambda_T} \to T(\overline{\Lambda_T})$. This function sends λ-trees to their respective subtrees.

Definition 4.1.2. The map unfold is the map

$$\mathsf{unfold} : \overline{\Lambda_T} \to V + (V \times \overline{\Lambda_T}) + (\overline{\Lambda_T} \times \overline{\Lambda_T})$$

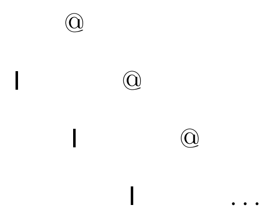

Figure 4.1.1. The infinite term I^ω

defined as

$$\mathsf{unfold}\,(x) = \kappa_1(x) \tag{4.1.2a}$$

$$\mathsf{unfold}\left(\begin{array}{ccc} & \lambda & \\ / & & \backslash \\ x & & M \end{array}\right) = \kappa_2(x, M) \tag{4.1.2b}$$

$$\mathsf{unfold}\left(\begin{array}{ccc} & @ & \\ / & & \backslash \\ M & & N \end{array}\right) = \kappa_3(M, N) \tag{4.1.2c}$$

Example 4.1.3. As an example let us look at the unfolding of the term $K = \lambda xy.x$:

$$\mathsf{unfold}\left(\begin{array}{ccccc} & \lambda & & & \\ / & & \backslash & & \\ x & & & \lambda & \\ & & / & & \backslash \\ & & y & & x \end{array}\right) = \kappa_2\left(x, \begin{array}{ccc} & \lambda & \\ / & & \backslash \\ y & & x \end{array}\right)$$

$$\mathsf{unfold}\left(\begin{array}{ccc} & \lambda & \\ / & & \backslash \\ y & & x \end{array}\right) = \kappa_2(y, x)$$

$$\mathsf{unfold}\,(x) = \kappa_1(x)$$

Please note the last equation; a difference is made between the λ-*tree* x, which is an element of $\overline{\Lambda_T}$, and the *variable* x, which is an element of V.

Example 4.1.4. Let us look at another example: the unfolding of the infinite term $\mathsf{I}^\omega = (\mathsf{I}(\mathsf{I}(\mathsf{I}(\ldots))))$. It is shown in Figure 4.1.1. The unfolding looks like:

$$\mathsf{unfold}(\mathsf{I}^\omega) = \kappa_2(\mathsf{I}, \mathsf{I}^\omega)$$

The functor T, the carrier $\overline{\Lambda_T}$ and the map $\mathsf{unfold} : \overline{\Lambda_T} \to T(\overline{\Lambda_T})$ together give us the T-coalgebra $(\overline{\Lambda_T}, \mathsf{unfold})$. Now we only need to prove that this coalgebra is final, which will allow us to give coinductive definitions and proofs about this coalgebra.

Theorem 4.1.5. *The T-coalgebra $(\overline{\Lambda_T}, \mathsf{unfold})$ is a final coalgebra.*

Proof. In order to prove this, we need to take an arbitrary T-coalgebra (U, c) and show that:

$$\begin{tikzcd}
U \arrow[r, "h"] \arrow[d, "c"'] & \overline{\Lambda_T} \arrow[d, "\mathsf{unfold}", "\cong"'] \\
T(U) \arrow[r, "T(h)"] & T(\overline{\Lambda_T})
\end{tikzcd}$$

Figure 4.1.2. The diagram of $\mathsf{unfold} \circ h = T(h) \circ c$

existence: There exists a homomorphism of coalgebras $h : U \to \overline{\Lambda_T}$ that satisfies the equation $\mathsf{unfold} \circ h = T(h) \circ c$, and thus makes Diagram 4.1.2 commute.

uniqueness: This homomorphism is the only homomorphism satisfying this equation (up to isomorphism of the coalgebras).

We begin with the proof of existence by introducing the function

$$h : U \to \overline{\Lambda_T}$$

defined by

$$h(v) = \begin{cases} x & \text{if } c(v) = \kappa_1(x) \\ \begin{array}{ccc} & \lambda & \\ / & & \backslash \\ x & & h(w) \end{array} & \text{if } c(v) = \kappa_2(x, w) \\ \begin{array}{ccc} & @ & \\ / & & \backslash \\ h(w) & & h(w') \end{array} & \text{if } c(v) = \kappa_3(w, w') \end{cases} \tag{4.1.3}$$

It will be shown that this function is a homomorphism. This means that the function satisfies the following equation:

$$\mathsf{unfold} \circ h = T(h) \circ c \tag{4.1.4}$$

with

$$T(h) : T(U) \to T(\overline{\Lambda_T})$$

defined by

$$T(h) = [\mathsf{id}_V, \mathsf{id}_V \times h, h \times h] \tag{4.1.5}$$

To do this, we distinguish three cases:

1. $c(u) = \kappa_1(x)$

$$\begin{aligned}
\mathsf{unfold} \circ h \circ u &= \mathsf{unfold}(h(u)) & \\
&= \mathsf{unfold}(x) & \text{by (4.1.3)} \\
&= \kappa_1(x) & \text{by (4.1.2a)} \\
&= T(h) \circ \kappa_1(x) & \text{by (4.1.5)} \\
&= T(h) \circ c \circ u & \text{by assumption}
\end{aligned}$$

2. $c(u) = \kappa_2(x, w)$

$$\begin{aligned}
\mathsf{unfold} \circ h \circ u &= \mathsf{unfold}\left(\begin{array}{ccc} & \lambda & \\ & / \quad \backslash & \\ x & & h(w) \end{array}\right) & \text{by (4.1.3)} \\
&= \kappa_2(x, h(w)) & \text{by (4.1.2b)} \\
&= T(h) \circ \kappa_2(x, w) & \text{by (4.1.5)} \\
&= T(h) \circ c \circ u & \text{by assumption}
\end{aligned}$$

3. $c(u) = \kappa_3(w, w')$

$$\begin{aligned}
\mathsf{unfold} \circ h \circ u &= \mathsf{unfold}\left(\begin{array}{ccc} & @ & \\ & / \quad \backslash & \\ h(w) & & h(w') \end{array}\right) & \text{by (4.1.3)} \\
&= \kappa_3(h(w), h(w')) & \text{by (4.1.2c)} \\
&= T(h) \circ \kappa_3(w, w') & \text{by (4.1.5)} \\
&= T(h) \circ c \circ u & \text{by assumption}
\end{aligned}$$

This exhausts all possible cases, so we can conclude that h is indeed a T-coalgebra map from (U, c) to $(\overline{\Lambda_T}, \mathsf{unfold})$. In order to show that this homomorphism is unique let us assume that there is another function $g : U \to \overline{\Lambda_T}$ that makes diagram 4.1.2 commute. In other words:

$$\mathsf{unfold} \circ g = T(g) \circ c \tag{4.1.6}$$

where

$$T(g) = [\mathsf{id}_V, \mathsf{id}_V \times g, g \times g] \tag{4.1.7}$$

Again, consider these cases:

1. $c(u) = \kappa_1(x)$

$$\begin{aligned}
\mathsf{unfold} \circ g \circ u &= T(g) \circ c \circ u & \text{by (4.1.6)} \\
&= T(g) \circ \kappa_1(x) & \text{by assumption} \\
&= \kappa_1(x) & \text{by (4.1.7)}
\end{aligned}$$

2. $c(u) = \kappa_2(x, w)$

$$\begin{aligned}
\mathsf{unfold} \circ g \circ u &= T(g) \circ c \circ u && \text{by (4.1.6)} \\
&= T(g) \circ \kappa_2(x, w) && \text{by assumption} \\
&= \kappa_2(x, g(w)) && \text{by (4.1.7)}
\end{aligned}$$

3. $c(u) = \kappa_3(w, w')$

$$\begin{aligned}
\mathsf{unfold} \circ g \circ u &= T(g) \circ c \circ u && \text{by (4.1.6)} \\
&= T(g) \circ \kappa_3(w, w') && \text{by assumption} \\
&= \kappa_3(g(w), g(w')) && \text{by (4.1.7)}
\end{aligned}$$

As can be seen, g does exactly the same as h, so we can conclude that $g = h$. But since g was chosen as an arbitrary homomorphism, h must be the unique homomorphism that makes Diagram 4.1.2 commute. And so it can be concluded that the coalgebra $(\overline{\Lambda_T}, \mathsf{unfold})$ is final. □

Note. From now on this paper will use the λ-term notation instead of the λ-tree notation, when there can be no confusion. Examples of this conversion can be found above, for instance in Example 4.1.3.

4.2. Alpha-congruence

The next logical step would be to define alpha-congruence. Alpha-congruence is intuitively clear for both finite and infinite terms. However, defining it for infinite terms is not easy. First of all, it requires the set V of variables to be uncountable and it should always be possible to find an unused variable.

Secondly, alpha-conversion is usually defined in terms of substitutions of bound variables. The coalgebraic structure of lambda-terms is unfortunately not well suited for the identification of bound variables. When lambda-terms are destructed by the function unfold, the previous context gets lost.

Example 4.2.1. Let $f_\alpha : \overline{\Lambda_T} \to \overline{\Lambda_T}$ be a coinductively defined function which maps lambda-terms to other lambda-terms by changes of bound variables (as in Definition 1.2.4). This function would for example have the following observations for $\lambda x.x$:

$$\begin{aligned}
\mathsf{unfold}(f_\alpha(\lambda x.x)) &= \kappa_2(y, f_\alpha(x)) \\
\mathsf{unfold}(f_\alpha(x)) &= y
\end{aligned}$$

On the other hand, for the term x this function would have the following observation:

$$\mathsf{unfold}(f_\alpha(x)) = x$$

$$\begin{tikzcd}
\overline{\Lambda_T} \arrow[d, "\mathsf{unfold}"', "\cong"] & \overline{\Lambda_T} \times \overline{\Lambda_T} \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\alpha"] & \overline{\Lambda_T} \arrow[d, "\cong"', "\mathsf{unfold}"] \\
T(\overline{\Lambda_T}) & T(\overline{\Lambda_T} \times \overline{\Lambda_T}) \arrow[l, "T(\pi_1)"] \arrow[r, "T(\pi_2)"'] & T(\overline{\Lambda_T})
\end{tikzcd}$$

Figure 4.2.1. The hypothetical bisimulation relation α

The difference between these equations is a bit puzzling, but it all comes down to the fact that the second equation automagically remembers the fact that x was bound. Clearly, the function f_α as described above can not exist.

Another approach might be to define a bisimulation which relates alpha-congruent terms. However, it can be recalled from Definition 3.4.2 that such a bisimulation must have a structure α which makes Figure 4.2.1 commute. But since

$$T(\overline{\Lambda_T} \times \overline{\Lambda_T}) = V + (V \times \overline{\Lambda_T} \times \overline{\Lambda_T}) + ((\overline{\Lambda_T} \times \overline{\Lambda_T}) \times (\overline{\Lambda_T} \times \overline{\Lambda_T}))$$

such a bisimulation can not exist, because it requires all variables to be the same. And that is exactly what alpha-congruence is not.

There might be an elegant solution to the problem of alpha-congruence. However, in this thesis the problem will be ignored. It will be assumed that alpha-equivalent terms can be identified. Furthermore, the variable convention from Definition 1.3.2 is extended to infinite terms too.

While this approach might seem a bit shocking, the situation is not as bad as it seems. Section 4.5 will show that the coalgebraic approach described in this chapter holds equally well for De Bruijn terms.

4.3. Substitution

Now it is time to define some kind of substitution function subst which can substitute free variables within a λ-term by another λ-term. As has been explained in Chapter 3, this is usually done by first defining a new coalgebra on the functor T that has a suitable carrier and map, say the function f_{subst}. When that function has been defined, subst will be the unique homomorphism from f_{subst} to unfold.

The map f_{subst} should satisfy the following equations:

$$\begin{aligned}
\mathsf{unfold}(\mathsf{subst}(x, N, x)) &= \mathsf{unfold}(N) \\
\mathsf{unfold}(\mathsf{subst}(x, N, y)) &= \kappa_1(y) \\
\mathsf{unfold}(\mathsf{subst}(x, N, \lambda y.M)) &= \kappa_2(y, \mathsf{subst}(x, N, M)) \\
\mathsf{unfold}(\mathsf{subst}(x, N, M_1 M_2)) &= \kappa_3(\mathsf{subst}(x, N, M_1), \mathsf{subst}(x, N, M_2))
\end{aligned}$$

$$\begin{tikzcd}
V \times \overline{\Lambda_T} \times \overline{\Lambda_T} \arrow[r, "\mathsf{subst}"] \arrow[d, "f_{\mathsf{subst}}"'] & \overline{\Lambda_T} \arrow[d, "\mathsf{unfold}", "\cong"'] \\
T(\overline{\Lambda_T} + V \times \overline{\Lambda_T} \times \overline{\Lambda_T}) \arrow[r, "{T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}])}"'] & T(\overline{\Lambda_T})
\end{tikzcd}$$

Figure 4.3.1. The Flattening of f_{subst}

Note. The term $\mathsf{subst}(x, N, M)$ should be read as: "substitute all occurences of x by N in M". Usually we will write $M[x := N]$ instead of $\mathsf{subst}(x, N, M)$.

When we consider the first equation, we notice that in this case substitution is not defined in terms of itself, even though $\mathsf{unfold}(N)$ could contain other λ-terms. This is problematic because coinductively defined functions tend to traverse an entire data-structure, i.e. they are always defined in terms of themselves.

To solve this problem we employ the Flattening Lemma (Lemma 3.5.1). This lemma states that instead of having to define a function $f_{\mathsf{subst}} : V \times \overline{\Lambda_T} \times \overline{\Lambda_T} \to T(V \times \overline{\Lambda_T} \times \overline{\Lambda_T})$ we can also define the function $f_{\mathsf{subst}} : V \times \overline{\Lambda_T} \times \overline{\Lambda_T} \to T(\overline{\Lambda_T} + V \times \overline{\Lambda_T} \times \overline{\Lambda_T})$. The resulting homomorphism subst is unique by this lemma. The application of the flattening lemma to this problem can be seen in Diagram 4.3.1.

Definition 4.3.1. f_{subst} is defined as follows:

$$\begin{aligned}
f_{\mathsf{subst}}(x, y, x) &= \kappa_1(y) \\
f_{\mathsf{subst}}(x, \lambda z.N, x) &= \kappa_2(z, \kappa_1(N)) \\
f_{\mathsf{subst}}(x, N_1N_2, x) &= \kappa_3(\kappa_1(N_1), \kappa_1(N_2)) \\
f_{\mathsf{subst}}(x, N, y) &= \kappa_1(y) \\
f_{\mathsf{subst}}(x, N, \lambda y.M) &= \kappa_2(y, \kappa_2(x, N, M)) \\
f_{\mathsf{subst}}(x, N, M_1M_2) &= \kappa_3(\kappa_2(x, N, M_1), \kappa_2(x, N, M_2))
\end{aligned}$$

Note. The first three equations of this definition essentially say that $f_{\mathsf{subst}}(x, N, x) = N$.

Proposition 4.3.2. *The function* subst *that arises by the Flattening Lemma as the unique homomorphism from* $V \times \overline{\Lambda_T} \times \overline{\Lambda_T}$ *to* $\overline{\Lambda_T}$ *is the function* subst *as described above.*

Proof. By the Flattening Lemma, there is a function subst that satisfies:

$$\mathsf{unfold} \circ \mathsf{subst} = T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \tag{4.3.1}$$

Appendix B contains the proof that subst has the desired properties as described at the beginning of this section. □

4.4. Beta-reduction

Now that the λ-terms have been described and the notion of substitution is clear, we would like to finally do some useful things with our terms. However, we have not yet succeeded in giving a satisfactory definition of beta-reduction.

For now, only the function reduce will be defined. The complete head reduction map reduce is a map that walks down a λ-tree and reduces every redex it encounters (but does not necessarily reduce to a normal form, even if one exists). It satisfies the following equations:

$$\begin{aligned}
\mathsf{unfold}(\mathsf{reduce}(x)) &= \kappa_1(x) \\
\mathsf{unfold}(\mathsf{reduce}(\lambda x.M)) &= \kappa_2(x, \mathsf{reduce}(M)) \\
\mathsf{unfold}(\mathsf{reduce}((\lambda x.M)N)) &= \begin{cases} \kappa_1(y) & \text{if } M[x:=N]=y \\ \kappa_2(y, \mathsf{reduce}(M')) & \text{if } M[x:=N]=\lambda y.M' \\ \kappa_3(\mathsf{reduce}(M_1), \mathsf{reduce}(M_2)) & \text{if } M[x:=N]=M_1M_2 \end{cases} \\
\mathsf{unfold}(\mathsf{reduce}(MN)) &= \kappa_3(\mathsf{reduce}(M), \mathsf{reduce}(N))
\end{aligned}$$

As usual, we define this function coalgebraically by creating a T-coalgebra $(f_{\mathsf{reduce}}, \overline{\Lambda_T})$ and show that reduce is the unique homomorphism of coalgebras from this coalgebra to our final T-coalgebra. Unfortunately, this coalgebra is not easy to define; the third equation maps to different parts of $T(\overline{\Lambda_T})$. That means that the following definition of f_{reduce} is quite complex.

Definition 4.4.1 (The function f_{reduce}). The function

$$f_{\mathsf{reduce}} : \overline{\Lambda_T} \to T(\overline{\Lambda_T})$$

is defined by

$$\begin{aligned}
f_{\mathsf{reduce}}(x) &= \kappa_1(x) \\
f_{\mathsf{reduce}}(\lambda x.M) &= \kappa_2(x, M) \\
f_{\mathsf{reduce}}((\lambda x.x)y) &= \kappa_1(y) \\
f_{\mathsf{reduce}}((\lambda x.x)\lambda y.M) &= \kappa_2(y, M) \\
f_{\mathsf{reduce}}((\lambda x.x)(MN)) &= \kappa_3(M, N) \\
f_{\mathsf{reduce}}((\lambda x.y)N) &= \kappa_1(y) \\
f_{\mathsf{reduce}}((\lambda x.\lambda y.M)N) &= \kappa_2(y, M[x := N]) \\
f_{\mathsf{reduce}}((\lambda x.M_1M_2)N) &= \kappa_3(M_1[x := N], M_2[x := N]) f_{\mathsf{reduce}}(MN) \quad = \kappa_3(M, N)
\end{aligned}$$

Proposition 4.4.2. *The map* reduce *that arises as the unique homomorphism from* $(\overline{\Lambda_T}, f_{\mathsf{reduce}})$ *to* $(\overline{\Lambda_T}, \mathsf{unfold})$ *has the desired properties.*

Proof. This proposition is proven by case distinction in Appendix B. □

4.5. De Bruijn Trees

As promised in section 4.2 we will now show some related results for De Bruijn trees. Most of these results will be given without extensive proofs.

To model De Bruijn trees, we consider the functor

$$DB(X) = \mathbb{N} + X + X \times X$$

We equip this functor with a function $\mathsf{unfold}_{DB} : \overline{\Lambda_{DB}} \to T(\overline{\Lambda_{DB}})$, which is defined as:

$$\begin{aligned}
\mathsf{unfold}_{DB}(n) &= \kappa_1(n) \\
\mathsf{unfold}_{DB}(\lambda M) &= \kappa_2(M) \\
\mathsf{unfold}_{DB}(MN) &= \kappa_3(M, N)
\end{aligned}$$

Theorem 4.5.1. *The DB-coalgebra* $(\overline{\Lambda_{DB}}, \mathsf{unfold}_{DB})$ *is a final coalgebra.*

Proof. As the proof of theorem 4.1.5. □

In order to do useful things with our De Bruijn trees, we need the notion of renaming. This function rename should have the following observations:

$$\begin{aligned}
\mathsf{unfold}_{DB}(\mathsf{rename}(m, i, j)) &= \begin{cases} \kappa_1(j) & \text{if } j < i \\ \kappa_1(j + m - 1) & \text{if } j \geq i \end{cases} \\
\mathsf{unfold}_{DB}(\mathsf{rename}(m, i, \lambda N)) &= \kappa_2(\mathsf{rename}(N)) \\
\mathsf{unfold}_{DB}(\mathsf{rename}((m, i, N_1 N_2)) &= \kappa_3(\mathsf{rename}(m, i, N_1), \mathsf{rename}(m, i, N_2))
\end{aligned}$$

We induce the function rename as usual by means of a *DB*-coalgebra structure $(\mathbb{N} \times \mathbb{N} \times \overline{\Lambda_{DB}}, f_{\mathsf{rename}})$, which is defined as follows:

$$\begin{aligned}
f_{\mathsf{rename}}(m, i, j) &= \begin{cases} \kappa_1(j) & \text{if } j < i \\ \kappa_1(j + m - 1) & \text{if } j \geq i \end{cases} \\
f_{\mathsf{rename}}(m, i, \lambda N) &= \kappa_2(m, i, N) \\
f_{\mathsf{rename}}(m, i, N_1 N_2) &= \kappa_3((m, i, N_1), (m, i, N_2))
\end{aligned}$$

Proposition 4.5.2. *The function that arises as the unique homomorphism from* f_{rename} *to* unfold_{DB} *is the function* rename *as defined above.*

Proof. Identical to previous proofs of coinductive definitions □

As with $\overline{\Lambda_T}$, this allows us to define the function subst_{DB} by creating the coalgebra $f_{\mathsf{subst},DB} : \mathbb{N} \times \overline{\Lambda_{DB}} \times \overline{\Lambda_{DB}} \to DB(\overline{\Lambda_{DB}} + \mathbb{N} \times \overline{\Lambda_{DB}} \times \overline{\Lambda_{DB}})$. In this case, $f_{\mathsf{subst},DB}$ is defined as follows:

$$f_{\mathsf{subst},DB}(m,N,n) = \begin{cases} \kappa_1(n) & \text{if } n < m \\ \kappa_1(n-1) & \text{if } n > m \\ \kappa_1(\mathsf{rename}(n,1,j)) & \text{if } n = m \\ & \text{and } N = j \\ \kappa_2(\kappa_1(\mathsf{rename}(n,1,N'))) & \text{if } n = m \\ & \text{and } N = \lambda N' \\ \kappa_3\begin{pmatrix} \kappa_1(\mathsf{rename}(n,1,N_1)), \\ \quad \kappa_1(\mathsf{rename}(n,1,N_2)) \end{pmatrix} & \begin{array}{l}\text{if } n = m \\ \text{and } N = N_1 N_2\end{array} \end{cases}$$

$$f_{\mathsf{subst},DB}(m,N,\lambda M) = \kappa_2(\kappa_2(m+1,N,M))$$
$$f_{\mathsf{subst},DB}(m,N,M_1M_2) = \kappa_3(\kappa_2(m,N,M_1),\kappa_2(m,n,M_2)))$$

Proposition 4.5.3. *The function that arises as the unique homomorphism from $f_{\mathsf{subst},DB}$ to unfold_{DB} is the substitution function subst_{DB}.*

Proof. As the proof of subst. □

As can be seen from this section, the results on ‘normal’ lambda-terms are easily extended to De Bruijn terms. This might indicate that there exist natural transformations from T to DB and vice versa. If that is true, it would be possible to recognize alpha-congruent terms by transforming them to De Bruijn terms and showing that the resulting terms are bisimilar. However, natural transformations are beyond the scope of this paper too.

4.6. References

While this chapter consists of original ideas[1], it has of course been influenced by previous works.

Modelling infinite lambda-terms as infinite trees has been explored in the paper “Infinitary Lambda Calculus” by J. W. Klop et al. [**9**]; in this paper the trees are examined using positions and metrics on these positions. Substitutions have been explored in “Parametric Corecursion” by L. Moss[**13**]. The papers on coinduction by Jacobs and Rutten ([**5**], [**17**] and [**8**]) have of course been extremely valuable as well.

[1]Any similarities with existing papers on this subject are coincidence and show that this approach is indeed a natural one

Aftermath

All of the things that you used to do
If it is done, it is done by you

(Rolling Stones, [**16**])

In this thesis I have explored some of the possibilities that coalgebras have to offer for infinitary lambda-calculus. While there are a lot of open issues, the first results seem promising.

It has turned out that the notion of infinite trees is easily translated to a coalgebraic structure. Using infinite trees has the advantage that they are intuitively clear. Furthermore, manipulations on these trees are very similar to those on the finite lambda-calculus.

Of course, the absence of alpha-congruence is a big problem. However, it is probably not unsolvable. As has been suggested, natural transformations from $(\overline{\Lambda_T}, \mathsf{unfold})$ to $(\overline{\Lambda_{DB}}, \mathsf{unfold}_{DB})$ probably exist; if so, they can be used to identify alpha-congruent terms.

Apart from these observations, this thesis has been an interesting exercise in the use of coalgebras. This holds especially for the definition of substitution, since such definitions do not occur in the 'standard' example of infinite lists.

Once more, my gratitude goes out to everyone who helped me in writing this thesis.

Yigal Duppen
Amsterdam, August 2000

Appendix A

Symbols used in this Paper

In this chapter the contexts of all symbols in this paper are given. Subscripts and primes are not given, so when looking for the context of x' or x_n, look at the context of x.

Functions and relations have long names when it was believed that this would improve readability. Such functions and relations are all in $\mathsf{sans\ serif}$ font. By convention, functions start with a lowercase character while relations start with an uppercase character.

Symbol	Context
a	arbitrary constructor of an algebra
c	arbitrary structure of a coalgebra
f, g, k, l	arbitrary functions
h	an arbitrary homomorphism
i, j, m, n	arbitrary numbers
v, w	variables occurring in U
x, y, z	variables occurring in a λ-term
A, B, C, D	objects in a coalgebra
F, G, H	arbitrary functors
L, M, N	arbitrary λ-terms
P	arbitrary product in a coalgebra
S	arbitrary sum in a coalgebra
T	the lambda tree functor
U, V, W	arbitrary carriers of a coalgebra
X	arbitrary object in a coalgebra
Z	final object in a coalgebra
$\mathcal{C}$, $\mathcal{D}$	arbitrary categories
$\mathcal{F}$	the category of F-coalgebras
$\mathcal{S}$	the category of sets
Λ	the set of all finite λ-terms
Λ_T	the set of all finite λ-trees
$\overline{\Lambda_T}$	the set of all finite and infinite λ-trees

Appendix B

Proofs of Various Theorems and Propositions

B.1. The Proof of Proposition 4.3.2

In order to prove that subst has the desired observations, six cases are distinguished for $\mathsf{subst}(x, N, M)$ (or $M[x := N]$):

1. $M = x$

1a. $N = z$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, z, x) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, z, x) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_1(z) \\
&= \kappa_1(z) \\
&= \mathsf{unfold}(x)
\end{aligned}$$

1b. $N = \lambda z.N_1$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, \lambda z.N_1, x) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, \lambda z.N_1, x) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_2(z, \kappa_1(N_1)) \\
&= \kappa_2(z, [\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}] \circ \kappa_1(N_1)) \\
&= \kappa_2(z, N_1) \\
&= \mathsf{unfold}(\lambda z.N_1)
\end{aligned}$$

1c. $N = N_1 N_2$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, N_1 N_2, x) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, N_1 N_2, x) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_3(\kappa_1(N_1), \kappa_1(N_2)) \\
&= \kappa_3(N_1, N_2) \\
&= \mathsf{unfold}(N_1 N_2)
\end{aligned}$$

2. $M = y$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, N, y) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, N, y) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_1(y) \\
&= \kappa_1(y)
\end{aligned}$$

3. $M = \lambda y.M_1$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, N, \lambda y.M_1) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, N, \lambda y.M_1) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_2(y, \kappa_2(x, N, M_1)) \\
&= \kappa_2(y, [\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}] \circ \kappa_2(x, N, M_1)) \\
&= \kappa_2(y, \mathsf{subst}(x, N, M_1))
\end{aligned}$$

4. $M = M_1 M_2$

$$\begin{aligned}
\mathsf{unfold} \circ \mathsf{subst} \circ (x, N, M_1 M_2) &= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ f_{\mathsf{subst}} \circ (x, N, M_1 M_2) \\
&= T([\mathsf{id}_{\overline{\Lambda_T}}, \mathsf{subst}]) \circ \kappa_3(\kappa_2(x, N, M_1), \kappa_2(x, N, M_2)) \\
&= \kappa_3(\mathsf{subst}(x, N, M_1), \mathsf{subst}(x, N, M_2))
\end{aligned}$$

□

B.2. The Proof of Proposition 4.4.2

Proving that reduce has the desired properties proceeds by case distinction:

1.

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}(x)) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}(x))\\ &= T(\mathsf{reduce})(\kappa_1(x))\\ &= \kappa_1(x)\end{aligned}$$

2.

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}(\lambda x.M)) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}(\lambda x.M))\\ &= T(\mathsf{reduce})(\kappa_2(x, M))\\ &= \kappa_2(x, \mathsf{reduce}(M))\end{aligned}$$

$3a$. $(\lambda x.M)N = y$

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}((\lambda x.x)y) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda x.x)y))\\ &= T(\mathsf{reduce})(\kappa_1(y))\\ &= \kappa_1(y)\\ \mathsf{unfold}(\mathsf{reduce}((\lambda x.y)N) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda x.y)N))\\ &= T(\mathsf{reduce})(\kappa_1(y))\\ &= \kappa_1(y)\end{aligned}$$

$3b$. $(\lambda x.M)N = \lambda y.M' \quad M[x := N] = M'$

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}((\lambda x.x)\lambda y.M') &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda x.x)\lambda y.M')\\ &= T(\mathsf{reduce})(\kappa_2(y, M'))\\ &= \kappa_2(y, \mathsf{reduce}(M'))\\ \mathsf{unfold}(\mathsf{reduce}((\lambda xy.M)N) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda xy.M)N)\\ &= T(\mathsf{reduce})(\kappa_2(y, M[x := N])\\ &= T(\mathsf{reduce})(\kappa_2(y, M')\\ &= \kappa_2(y, \mathsf{reduce}(M'))\end{aligned}$$

$3c$. $(\lambda x.M)N = M_1M_2 \quad (M'M'')[x := N] = M_1M_2$

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}((\lambda x.x)M_1M_2) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda x.x)M_1M_2)\\ &= T(\mathsf{reduce})(\kappa_3(M_1, M_2))\\ &= \kappa_3(\mathsf{reduce}(M_1), \mathsf{reduce}(M_2))\\ \mathsf{unfold}(\mathsf{reduce}((\lambda x.M'M'')N) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}((\lambda x.M'M'')N)\\ &= T(\mathsf{reduce})(\kappa_3(M_1, M_2))\\ &= \kappa_3(\mathsf{reduce}(M_1), \mathsf{reduce}(M_2))\end{aligned}$$

4.

$$\begin{aligned}\mathsf{unfold}(\mathsf{reduce}(M_1M_2)) &= T(\mathsf{reduce})(f_{\mathsf{reduce}}(M_1M_2))\\ &= T(\mathsf{reduce})(\kappa_3(M_1, M_2))\\ &= \kappa_3(\mathsf{reduce}(M_1), \mathsf{reduce}(M_2))\end{aligned}$$

□

Bibliography

[1] Z. M. Ariola, J. W. Klop, *Cyclic Lambda Graph Rewriting*, Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 416-425, IEEE Computer Society Press, Paris, 1994

[2] H. P. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, Studies in Logic and the Foundation of Mathematics, vol. 103, North-Holland Publishing Company, 1984

[3] P. -L. Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming*, 2nd edition, Birkhäuser, Boston, 1993

[4] Chris Hankin, *Lambda Calculi, a Guide for Computer Scientists*, Clarendon Press, Oxford, 1994

[5] Bart Jacobs, *Exercises in Coalgebraic Specification*, `http://www.cs.kun.nl/~bart/PAPERS`

[6] Eric S. Raymond et al., *The Jargon File*, version 4.2.0, 31 Jan 2000, `http://www.tuxedo.org`

[7] F. William Lawvere, Stephen H. Schanuel, *Conceptual Mathematics, a First Introduction to Categories*, 2nd printing, Cambridge University Press, 1997

[8] Bart Jacobs, Jan Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, `http://www.cs.kun.nl/~bart/PAPERS/`

[9] J. R. Kennaway, J. W. Klop, M. R. Sleep, F. -J. de Vries, *Infinitary Lambda Calculus*, Report CS-R9535, CWI, 1995

[10] J. Lambek, P. J. Scott, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986

[11] Marina Lenisa, *Semantic Techniques for Deriving Coinductive Characterizations of Observational Equivalences for λ-calculi*, `http://www.dimi.uniud.it/~lenisa/Papers/list-papers.html`

[12] Marina Lenisa, *From Set-theoretic Coinduction to Coalgebraic Coinduction: Some Results, Some Problems*, `http://www.dimi.uniud.it/~lenisa/Papers/list-papers.html`

[13] Lawrence S. Moss, *Parametric Corecursion*, `http://math.indiana.edu/home/moss/articles.html`

[14] Mathematical Quotations Server,
`http://math.furman.edu/~mwoodard/mqs/mquot.shtml`

[15] Benjamin C. Pierce, *Basic Category Theory for Computer Scientists*, 2nd printing, Massachusets Institution of Technology, 1993

[16] The Rolling Stones, *Aftermath*, Decca, 1996

[17] J. J. M. M. Rutten, *Universal Coalgebra: a Theory of Systems*,
`http://www.cwi.nl/~janr/papers/`

[18] *The Silly Stuff Nexus*,
`http://www.animenetwork.com/sillystuff`

[19] Masako Takahashi, *Parallel Reductions in λ-Calculus*, Revised version, Information and Computation, 118(1):120-127, 1995