Lectures on Semantics: The initial algebra and final coalgebra perspectives

Peter Aczel
Departments of Mathematics and Computer Science
Manchester University,
Manchester, M13 9PL, UK

Abstract. These lectures give a non-standard introduction, for computer science students, to the mathematical semantics of formal languages. We do not attempt to give a balanced treatment, but instead focus on some key general ideas, illustrated with simple examples. The ideas are formulated using some elementary category theoretic notions. All the required category theory is introduced in the lectures. In addition to the familiar initial algebra approach to syntax and semantics we examine the less familiar final coalgebra approach to operational semantics. Our treatment of formal semantics is intended to complement a more standard introduction.

Keywords. semantics, formal language, category theory, category, functor, initial algebra, final coalgebra, operational semantics, CCS

Introduction

These lectures are primarily intended as an introduction, for computer science students, to some aspects of the mathematical semantics of formal languages. The lectures do not try to survey the field in any sense but focus on a cluster of ideas to do with the viewpoint that syntax and semantics are somehow dual to each other. In order to express this viewpoint we need to use some basic notions from category theory. So a secondary aim of these lectures is to give an introduction to those notions of category theory that are needed for the primary purpose.

The lectures only give a partial introduction to semantics and elementary category theory. But I believe that the combination of an introduction to the two topics is productive of a useful complement to the more standard introductions. For those of you who already have some familiarity with both topics I hope that there will still be something novel to interest you.

Four lectures were given at the summer school. In this paper I have tried to keep roughly to the organisation of the lectures, with one section per lecture. But it seemed worthwhile in parts to add extra details to those given in the lectures. The first lecture provides an introduction to mathematical semantics. It introduces some very simple examples of abstract syntax and associated notions of structure, which are used to illustrate the abstract definitions used in later lectures. The second lecture introduces the notion of a category, the illustrative examples being categories of structures, as well as the category of sets. The third lecture introduces the central ideas of these lectures: the notion of a bounded standard endo functor on the category of sets, categories of algebras and coalgebras for an endo functor, initial algebras and final coalgebras. There are two important general theorems on the existence of initial algebras and final coalgebras. There is an outline proof of the Initial Algebra Theorem, but not of the Final Coalgebra Theorem. Instead, in the last part of lecture three final coalgebras are described related to the four running examples of abstract syntax initiated in the first lecture. The fourth and final lecture is concerned with final universes of processes. The lecture discusses three kinds of computational paradigm, the traditional non-interactive deterministic paradigm, two interactive deterministic paradigms and an interactive non-deterministic paradigm. In each case a bounded standard endo functor is used and a final coalgebra for it gives the final universe of processes for the paradigm. The interactive non-deterministic paradigm that I use is the paradigm associated with CCS up to strong bisimulation, but there is only time to give the briefest sketch of this.

In these lectures very simple and familiar examples are used to illustrate the ideas. There is a close connection between universes of syntactic expressions given by abstract syntax and inductive data types. The examples may be viewed as inductive data types, but I present them using the terminology of formal languages. Dually the discussion of final coalgebras could be phrased as a discussion of coinductive data types.

1 Mathematical Semantics

1.1 General Introduction to Semantics

The distinction between the subjects of syntax and semantics has its origin in the study of natural languages. Syntax is concerned with the structural properties of the forms of expression of a language, usually divorced from any attention to their meaning. In semantics attention is focused on the meaning of expressions. Natural languages occur 'in nature' and have generally evolved over long periods of time. Their syntax and semantics are only given implicitly and it is necessary to observe the languages in use in order to develop an understanding of their syntax and semantics. This is part of the subject matter of linguistics.

By contrast the formal languages, that are the concern of these lectures, are explicit constructions and usually have an explicitly specified syntax and ideally an explicit semantics. The paradigm examples of formal languages arose in logic. They are the classical Propositional and Predicate calculi and their myriad variations and extensions. Formal languages are also used in linguistics where they can play a theoretical role as models of fragments of natural language in the same kind of way that mathematical models are used in the physical sciences.

In computer science the most immediate examples of formal languages are the programming languages. The general idea of a formal language has a much wider application, in computer science, than simply to programming languages. But for our purposes here it will be enough to consider them for a moment.

Historically, the designers of programming languages did not necessarily describe their languages by giving completely precise explicit syntax and semantics. They could construct a compiler (or interpreter) for a particular machine and write a manual that gives examples of programs in the language that could be run on the machine. A programmer might learn the language by reading the manual, by the trial and error writing of programs and the help of experts. A programming language that is available to us in that way is somewhat like a natural language. In order to gain a deeper understanding of the language, it may be necessary to observe the language closely and construct an explicit syntax and semantics that is something more abstract and theoretical than the compiler running on a particular machine.

There are many reasons why a deeper more abstract understanding of programming languages is desirable. But I will not go into them here. Suffice it to say that, because of the progress that has been made in the mathematical study of syntax and semantics, it is now possible in principle to present the design of a programming language as a mathematical object consisting of a precise syntax and semantics.

Formal Languages

A **formal language** has a **syntax** that specifies the syntactic categories of expressions of the language and also a **semantics** that specifies some possible **interpretations** of the syntax that give meanings to the expressions of the language. Sometimes there is a distinguished interpretation that might be called the **intended** or **standard** interpretation, the others being **unintended** or **non-standard**. A formal language may originally only have the single intended interpretation and only later other possible interpretations may be considered. It can also happen that a formal language is provided with no semantics, or rather the semantics is empty, providing no interpretations. The uninterpreted **formal systems**, studied by logicians, are examples of these.

Instead of focussing on a single formal language, with a single syntax and possibly many interpretations, it can be interesting to focus on a uniform approach to the intended interpretation of a **family of formal languages**. A main theme of these lectures is that the second idea can be viewed as dual to the first in a precise category theoretic sense.

There are two dual topics that we wish to examine, using some ideas of category theory, each requiring the choice of an endo functor:-

1. The abstract specification of a formal language by focussing on the abstract representation of the syntax as an initial object of the category of algebras for an endo functor, with the interpretations also represented as objects of the category and the meaning functions as arrows.

2. The specification of a particular uniform approach to representing the intended interpretation of each of a family of formal languages, as a final object of the category of coalgebras for an endo functor, with the syntax of each language also represented as objects of the category and the meaning functions as arrows.

Of course the above topics will be unclear to the reader who is not yet familiar with the category theoretic terminology. But all the terminology will be introduced here in these lectures.

The topic of these lectures is **mathematical semantics**. It is possible to be concerned with a **direct semantics** for a formal language. For example one may wish to set up a formal language for the foundations of mathematics. In that case one may want to have a direct pre-mathematical semantics that would be quite different from a mathematical semantics, which is essentially a translation of the syntax of the formal language into our mathematical language. Similarily one can distinguish between our direct understanding of our native natural language and the understanding of a foreign language that we might get via translation.

1.2 Examples of Syntax and Semantics

In these lectures we focus on four very simple examples of formal languages with syntax and semantics. They are unrealistically simple so that the general ideas can be seen uncluttered with the details of more realistic examples. In each case we give an abstract syntax determining a set E of expressions, we give an associated notion of mathematical structure and for each such structure, $\mathcal{A} = (A, \ldots)$, a **meaning function** $[[\ ]]^{\mathcal{A}} : E \to A$. The first component A of the structure is a set, called the **underlying set** of the structure. So the meaning function associates a value in the underlying set to each expression. The mathematical structures are the interpretations of the formal language. The meaning function is defined by a structural recursion following the way expressions are generated; i.e. the function is specified as the unique function satisfying the defining equations, one equation for each form of expression.

In the second and third examples we assume given a fixed set K. In the fourth example we assume given a signature consisting of a set Σ of **function symbols**, each $\sigma \in \Sigma$ having an associated natural number n_σ as its **arity**.

Peano Structures

A **Peano structure**, $\mathcal{A} = (A, a, f)$, consists of a set A together with $a \in A$ and $f : A \to A$. The set E of expressions for this example is given by the abstract syntax

$$e ::= \Box \mid e^{+}.$$

So $E = \{\Box, \Box^{+}, \Box^{++}, \ldots\}$. Given a Peano structure $\mathcal{A} = (A, a, f)$, the meaning function $[\![\ ]\!]^{\mathcal{A}}$ is the unique function $[\![\ ]\!] : E \to A$ such that

$$\begin{cases} [\![\Box]\!] & = a \\ [\![e^{+}]\!] & = f([\![e]\!]) \quad (e \in E) \end{cases}$$

List structures

Given a set K, a **List structure over** K, $\mathcal{A} = (A, a, f)$, consists of a set A together with $a \in A$ and $f : K \times A \to A$. This time the set E of expressions is given by the abstract syntax

$$e ::= [] \mid k : e \qquad\qquad (k \in K)$$

A typical element of E has the form $k_1 : \cdots : k_n : []$, where $k_1, \ldots, k_n$ are elements of K. Given a List structure $\mathcal{A} = (A, a, f)$, the meaning function $[\![\ ]\!]^{\mathcal{A}}$ is the unique function $[\![\ ]\!] : E \to A$ such that

$$\begin{cases} [\![[]]\!] & = a \\ [\![k : e]\!] & = f(k, [\![e]\!]) \quad (k \in K, e \in E) \end{cases}$$

Lisp structures

Given a set K, a **Lisp structure over** K, $\mathcal{A} = (A, a, f)$, consists of a set A together with $a : K \to A$ and $f : A \times A \to A$. The set E of expressions is given by the abstract syntax

$$e ::= k \mid e \cdot e \qquad\qquad (k \in K)$$

Given a Lisp structure $\mathcal{A} = (A, a, f)$, the meaning function $[\![\ ]\!]^{\mathcal{A}}$ is the unique function $[\![\ ]\!] : E \to A$ such that

$$\begin{cases} [\![k]\!] & = a(k) \quad (k \in K) \\ [\![e_1 \cdot e_2]\!] & = f([\![e_1]\!], [\![e_2]\!]) \quad (e_1, e_2 \in E) \end{cases}$$

Σ-structures

Given a signature Σ, a **Σ-structure** $\mathcal{A} = (A, \sigma^{\mathcal{A}})_{\sigma \in \Sigma}$ consists of a set A together with $\sigma^{\mathcal{A}} : A^{n_\sigma} \to A$ for each $\sigma \in \Sigma$. When $n_\sigma = 0$ then A^{n_σ} is a singleton set consisting of the empty tuple so that $\sigma^{\mathcal{A}}$ is just picking out the element $\sigma^{\mathcal{A}}()$ of A. It is standard to identify $\sigma^{\mathcal{A}}$ with this element The set E of expressions is given by the abstract syntax

$$e ::= \sigma\, \overbrace{e \cdots e}^{n_\sigma} \qquad\qquad (\sigma \in \Sigma).$$

Given a Σ-structure $\mathcal{A} = (A, \sigma^{\mathcal{A}})_{\sigma \in \Sigma}$, the meaning function $[\![\;]\!]^{\mathcal{A}}$ is the unique function $[\![\;]\!] : E \to A$ such that

$$[\![\sigma e_1 \cdots e_{n_\sigma}]\!] = \sigma^{\mathcal{A}}([\![e_1]\!], \ldots, [\![e_{n_\sigma}]\!]) \quad (\sigma \in \Sigma, e_1 \ldots, e_{n_\sigma} \in E).$$

Exercise 1.1 *Show that the previous three examples can be treated as special cases of this by choosing*

1. $\Sigma = \{\Box, +\}$, *with* $n_\Box = 0$, $n_+ = 1$.
2. $\Sigma = \{[]\} \cup K$, *assuming* $[]$ *is not in* K, *with* $n_{[]} = 0$, $n_k = 1$ *for* $k \in K$.
3. $\Sigma = K \cup \{\cdot\}$, *assuming* $\cdot$ *is not in* K, *with* $n_k = 0$ *for* $k \in K$ *and* $n_\cdot = 2$.

1.3 On structural induction and structural recursion

For the convenience of readers unfamiliar with the structural induction and structural recursion that can be used with abstract syntax we review the ideas in connection with our fourth example, the Σ-structures. Those readers should treat this interlude as an extended exercise to fill in the details.

A syntax equation $e ::= \cdots$, for a set E of expressions e, specifies an inductive definition of E. In our fourth example E is inductively defined as the smallest set such that for each $\sigma \in \Sigma$,

$$e_1, \ldots, e_{n_\sigma} \in E \quad \Rightarrow \quad \sigma e_1 \cdots e_{n_\sigma} \in E.$$

This inductive definition can be reformulated as the explicit definition:

$$E = E^0 \cup E^1 \cup \cdots,$$

where the E^k are defined primitive recursively by the equations $E^0 = \emptyset$ and for $k \in \mathbb{N}$

$$E^{k+1} = \{\sigma e_1 \cdots e_{n_\sigma} \mid \sigma \in \Sigma, e_1, \ldots, e_{n_\sigma} \in E^k\}.$$

Note that by mathematical induction $E^0 \subseteq E^1 \subseteq E^2 \subseteq \cdots$.

The stipulation that the syntax is abstract means that expressions have an unambiguous form. So each expression in E must have the form $\sigma e_1 \cdots e_{n_\sigma}$ for uniquely determined $\sigma \in \Sigma$ and $e_1, \ldots, e_{n_\sigma} \in E$. The particular concrete representation of expressions, say as strings or trees, is not important provided that the unambiguity condition holds. This unambiguity condition is essential in order to define the meaning function on E by structural recursion. Given a Σ-structure $\mathcal{A}$ the definition of the meaning function $[[\;]]^{\mathcal{A}}$ by structural recursion can be reformulated in terms of an ordinary primitive recursive definition of functions $[[\ ]]_k : E^k \rightarrow A$ for $k \in \mathbb{N}$ as the unique sequence of functions such that $[[\ ]]_0 : \emptyset \rightarrow A$ is the unique empty function and for $k \in \mathbb{N}$

$$[[\sigma e_1 \cdots e_{n_\sigma}]]_{k+1} = \sigma^{\mathcal{A}}([[e_1]]_k, \ldots, [[e_{n_\sigma}]]_k),$$

if $\sigma \in \Sigma$ and $e_1, \ldots, e_{n_\sigma} \in E^k$. Note that the unambiguity condition is used here, when defining $[[\ ]]_{k+1}$ in terms of $[[\ ]]_k$. Observing that, by mathematical induction on $k \in \mathbb{N}$, $[[e]]_{k+1} = [[e]]_k$ for $e \in E^k$, we can now define $[[e]]^{\mathcal{A}} = [[e]]_k$ for any $e \in E$, provided that $k \in \mathbb{N}$ is large enough so that $e \in E^k$.

1.4 Examples of Structures

Peano Structures

The standard example is $(\mathbb{N}, 0, s)$ where $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers and $s(n) = n + 1$ for $n \in \mathbb{N}$. Another example that will interest us later is $(\mathbb{N}^\infty, 0, s^\infty)$, where $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ and

$$s^\infty(n) = \begin{cases} s(n) & \text{if } n \in \mathbb{N} \\ \infty & \text{if } n = \infty \end{cases} .$$

Here ∞ is an object not in $\mathbb{N}$.

List Structures

We shall take $(K^*, \epsilon, cons)$ as our standard example, where K^* is the set of all (finite) strings $\rho = k_1 \cdots k_n$ of elements of K, ϵ is the empty string and

$$cons(k, \rho) = k k_1 \cdots k_n$$

if $k \in K$ and $\rho = k_1 \cdots k_n \in K^*$.

We shall also be interested in the List structure $(K^\infty, \epsilon, cons^\infty)$ of possibly infinite strings. Here $K^\infty = K^* \cup K^\omega$ where K^ω is the set of infinite strings $\rho = k_1 k_2 \cdots$ of elements of K and $cons^\infty(k, \rho) = k k_1 \cdots$ if $k \in K$ and $\rho = k_1 \cdots \in K^\infty$.

Lisp Structures

Given the set K, the standard example is the Lisp structure $(\mathbb{S}, atom, dot)$, where $\mathbb{S}$ is the set of Lisp S-expressions that are built up from atoms $atom(k)$ for $k \in K$ using a dotted pairing operation dot that associates with S-expressions M_1, M_2 an S-expression $M_1 \cdot M_2$.

We shall also be interested in a Lisp structure with possibly circular and infinite S-expressions. These can be viewed as certain kinds of binary trees whose leaves are labelled with elements of K. It will be useful to have a precise set theoretical definition of these trees. We start with a more general notion of S-**pretree**. These are defined to be pairs $M = (|M|, l_M)$ where $|M| \subseteq \{1,2\}^*$ and $l_M : |M| \to K \cup \{\cdot\}$. Here $\cdot$ is some object not in K. Let $\mathbb{PS}$ be the set of these pre-trees. If $k \in K$ let $atom(k)$ be the pre-tree M given by

$$\begin{cases} |M| & = \{\epsilon\} \\ l_M(\epsilon) & = k \end{cases}$$

If M_1, M_2 are pre-trees let $M_1 \cdot M_2$ be the pre-tree M given by

$$\begin{cases} |M| & = \{\epsilon\} \cup \{1\rho \mid \rho \in |M_1|\} \cup \{2\rho \mid \rho \in |M_2|\} \\ l_M(\epsilon) & = \cdot \\ l_M(1\rho) & = l_{M_1}(\rho) \text{ for } \rho \in |M_1| \\ l_M(2\rho) & = l_{M_2}(\rho) \text{ for } \rho \in |M_2| \end{cases}$$

We can now form a Lisp structure $(\mathbb{PS}, \cdots)$ of pretrees in the obvious way.

The S-**trees** are defined to be those S-pretrees M such that $\epsilon \in M$ and if $\rho \in \{1,2\}^*$, $i \in \{1,2\}$ then

$$\rho i \in |M| \Leftrightarrow \rho \in |M| \text{ and } l_M(\rho) = \cdot).$$

It turns out that we can now define $\mathbb{S}$ to be the set of finite S-trees; i.e. the set of those trees M where $|M|$ is a finite set. But we shall also be interested in the set $\mathbb{S}^\infty$ of all S-trees. Both these sets can be made into Lisp structures in the obvious way using *atom* and the dotted pair operation on trees.

Exercise 1.2 *Show that if M_1, M_2 are (finite) trees then so is $M_1 \cdot M_2$. Also show that any (finite) tree can be uniquely expressed in one of the forms $atom(k)$ for some $k \in K$ or $M_1 \cdot M_2$ for some (finite) trees M_1, M_2.*

Σ-structures

Our examples will be obtained by generalising the notion of S-pretree. We define a Σ-**pretree** to be a pair $M = (|M|, l_M)$ such that $M \subseteq \mathbb{N}^*$ and $l_M : |M| \to \Sigma$ and let $\mathbb{P}\Sigma$ be the set of all these pre-trees. If $\sigma \in \Sigma$ and $M_1, \ldots, M_{n_\sigma}$ are pre-trees then we define $\sigma(M_1, \ldots, M_{n_\sigma})$ to be the pretree M given by

$$\begin{cases} |M| & = \{\epsilon\} \cup \bigcup_{i=1,\ldots,n_\sigma} \{i\rho \mid \rho \in |M_i|\} \\ l_M(\epsilon) & = \sigma \\ l_M(i\rho) & = l_{M_i}(\rho) \text{ for } i \in \{1,\ldots,n_\sigma\},\ \rho \in |M_i|. \end{cases}$$

With this definition the set $\mathbb{P}\Sigma$ of σ-pretrees can be made into a Σ-structure. But we will be more interested in the substructure of Σ-trees. A pretree M is a Σ-**tree** if $\epsilon \in |M|$ and for $\rho \in \mathbb{N}^*$, $i \in \mathbb{N}$

$$\rho i \in |M| \Leftrightarrow \rho \in |M| \text{ and } i \in \{1, \ldots, n_\sigma\}, \text{ where } \sigma = l_M(\rho).$$

Exercise 1.3 *Show that if $\sigma \in \Sigma$ and $M_1, \ldots, M_{n_\sigma}$ are (finite) trees then so is $\sigma(M_1, \ldots, M_{n_\sigma})$. Show that for every (finite) tree M*

$$M = \sigma(M_1, \ldots, M_{n_\sigma})$$

for uniquely determined $\sigma \in \Sigma$ and (finite) trees $M_1, \ldots, M_{n_\sigma}$.

Let $\mathbb{T}\Sigma$ be the set of finite Σ-trees and $\mathbb{T}^\infty\Sigma$ be the set of all Σ-trees. We can now define the Σ-structures $(\mathbb{T}\Sigma, \cdots)$ and $(\mathbb{T}^\infty\Sigma, \cdots)$ in the obvious way.

1.5 Syntax as a structure

In each of our four examples of syntax and semantics we form a structure $\mathcal{E} = (E, \ldots)$ out of the set of expressions E. I call this the **syntax structure**.

Example 1 $\mathcal{E} = (E, \Box, s)$, where $s : E \to E$ is given by $s(e) = e^+$ for $e \in E$.

Example 2 $\mathcal{E} = (E, [], cons)$, where

$$cons : K \times E \to E$$

is given by $cons(k, e) = k : e$ for $k \in K$, $e \in E$.

Example 3 $\mathcal{E} = (E, atom, dot)$, where $atom : K \to E$ is given by $atom(k) = k$ for $k \in K$ and $dot : E \times E \to E$ is given by $dot(e_1, e_2) = e_1 \cdot e_2$ for $e_1, e_2 \in E$.

Example 4 $\mathcal{E} = (E, \sigma^{\mathcal{E}})_{\sigma \in \Sigma}$, where for each $\sigma \in \Sigma$ the operation $\sigma^{\mathcal{E}} : E^{n_\sigma} \to E$ is given by

$$\sigma^{\mathcal{E}}(e_1, \ldots, e_{n_\sigma}) = \sigma e_1 \cdots e_{n_\sigma}$$

for $e_1, \ldots e_{n_\sigma} \in E$.

2 Categories of Structures

2.1 Categories

In this subsection we introduce the notion of a category. Our initial motivating example is the category of sets and functions between sets.

Functions between Sets

If A, B are sets then we write $f : A \to B$ if f is a function from A to B. The set A is the **domain** of f and B is the **codomain** of f. For each $x \in A$ the result $f(x)$, of **applying** f to x, is in B. When the codomain coincides with the **range**, $\{f(x) \mid x \in A\}$, then f is **surjective**. But it need not be. So functions cannot be identified with their graphs, as is usual in set theory, but carry with them their codomain. This means that if $f : A \to B$ and $f' : A' \to B'$ then $f = f'$ iff $A = A'$, $B = B'$ and $f(x) = f'(x)$ for all $x \in A$.

If A is a set then the **identity on** A, id_A, is the unique function such that $id_A : A \to A$ and $id_A(x) = x$ for all $x \in A$.

If $f : A \to B$ and $g : B \to C$ then (g, f) is **composable** and their **composite**, $g \circ f$, is the unique function such that $g \circ f : A \to C$ and $(g \circ f)(x) = g(f(x))$ for all $x \in A$.

Note: If $f : A \to B$ then

$$f \circ id_A = id_B \circ f = f.$$

If also $g : B \to C$ and $h : C \to D$ then

$$(h \circ g) \circ f = h \circ (g \circ f).$$

The definition

Definition: 2.1 *A* **category** *consists of* **objects** *and* **arrows** *and four operations satisfying certain conditions given below. The four operations are*

1. *an assignment of an object, $dom(f)$, to each arrow f, called the* **domain** *of f,*

2. *an assignment of an object, $codom(f)$, to each arrow f, called the* **codomain** *of f,*

3. *an assignment of an arrow $g \circ f$, to each composable pair (g, f) of arrows, called the* **composite** *of (g, f),*

4. *an assignment of an arrow id_A to each object A, called the* **identity** *on A.*

In 3 a **composable pair** *of arrows is a pair of arrows, (g, f) such that $dom(g) = codom(f)$. The four operations are required to satisfy the following conditions.*

C-1 *If (g, f) is a composable pair of arrows then*

$$dom(g \circ f) = dom(f) \text{ and } codom(g \circ f) = codom(g).$$

C-2 *If f, g, h are arrows such that $(g, f), (h, g)$ are composable pairs then*

$$(h \circ g) \circ f = h \circ (g \circ f).$$

C-3 *If A is an object then*

$$dom(id_A) = codom(id_A) = A.$$

C-4 *If f is an arrow, with $A = dom(f)$ and $B = codom(f)$ then*

$$f \circ id_A = id_B \circ f = f.$$

When $dom(f) = A$ and $codom(f) = B$ then we write $f : A \rightarrow B$. When (g, f) is a composable pair, with $f : A \rightarrow B$ and $g : B \rightarrow C$ then we can write $A \xrightarrow{f} B \xrightarrow{g} C$ for the composite $g \circ f$.

The Category of Sets

Our first motivating example of a category, the category, **Set**, of sets can now be defined. The **objects** of **Set** are sets and the **arrows** of the category are the 'functions between sets' as discussed earlier. The four operations of domain, codomain, composition and identity have also been introduced and it should be clear that the conditions for a category hold.

2.2 Categories of structures

In this subsection we give further examples of categories based on the four examples of notions of structure given earlier. In each case the objects of the category are the structures and the arrows will be triples consisting of a pair of structures with a homomorphism between them; i.e. a function between the underlying sets of the structures that 'preserve structure'. Triples are needed, rather than the homomorphisms alone, so that the domain and codomain operations of the category can be defined. But in practise it is often convenient to blur the distinction between the arrow and its homomorphism component.

The category of Peano structures

The category **Pea** has the Peano structures as its objects. Given Peano structures $\mathcal{A} = (A, a, f)$ and $\mathcal{A}' = (A', a', f')$ a **homomorphism from $\mathcal{A}$ to $\mathcal{A}'$** is a function $\pi : A \rightarrow A'$ that 'preserves the structure'; i.e.

$$\begin{cases} \pi(a) & = a' \\ \pi(f(x)) & = f'(\pi(x)) \text{ for } x \in A \end{cases}$$

The arrows of **Pea** are the triples $(\mathcal{A}, \pi, \mathcal{A}')$ consisting of Peano structures with a homomorphism between them. It should be clear what the category operations are and also clear that the conditions for being a category hold. In the other three examples we will just describe the notion of homomorphism.

The category of List structures

Given a set K the category **List**(K) has the List structures over K as its objects and the notion of homomorphism that determines the arrows is given as follows. Given List structures $\mathcal{A} = (A, a, f)$ and $\mathcal{A}' = (A', a', f')$ over K a **homomorphism from $\mathcal{A}$ to $\mathcal{A}'$** is a function $\pi : A \to A'$ such that

$$\begin{cases} \pi(a) & = a' \\ \pi(f(k, x)) & = f'(k, \pi(x)) \text{ for } k \in K, x \in A \end{cases}$$

The category of Lisp structures

Given a set K the category $\mathbf{Lisp}(K)$ has the Lisp structures over K as its objects and the notion of homomorphism that determines the arrows is given as follows. Given Lisp structures $\mathcal{A} = (A, a, f)$ and $\mathcal{A}' = (A', a', f')$ over K a **homomorphism from $\mathcal{A}$ to $\mathcal{A}'$** is a function $\pi : A \to A'$ such that

$$\begin{cases} \pi(a(k)) & = a'(k) \text{ for } k \in K \\ \pi(f(x_1, x_2)) & = f'(\pi(x_1), \pi(x_2)) \text{ for } x_1, x_2 \in A \end{cases}$$

The category of Σ-structures

Given a signature Σ the category $\mathbf{Str(\Sigma)}$ has the Σ-structures as its objects and the notion of homomorphism that determines the arrows is given as follows. Given Σ-structures $\mathcal{A} = (A, \sigma^{\mathcal{A}})_{\sigma \in \Sigma}$ and $\mathcal{A}' = (A', \sigma^{\mathcal{A}'})_{\sigma \in \Sigma}$ a **homomorphism from $\mathcal{A}$ to $\mathcal{A}'$** is a function $\pi : A \to A'$ such that for $\sigma \in \Sigma$ and $x_1, \ldots, x_{n_\sigma} \in A$

$$\pi(\sigma^{\mathcal{A}}(x_1, \ldots, x_{n_\sigma})) = \sigma^{\mathcal{A}'}(\pi(x_1), \ldots, \pi(x_{n_\sigma}))$$

Exercise 2.2 *Show that, in each case above, the structures and homomorphisms between them do indeed determine a category.*

2.3 Syntax Structures are initial objects

We have now got four examples of categories of structures. In each of these categories we have defined a syntax structure $\mathcal{E}$. For each structure $\mathcal{A} = (A, \ldots)$ in the category there is a meaning function $[\![\ ]\!]^{\mathcal{E}} : E \to A$ that was defined by structural recursion on the form of the expressions in E; i.e. it was defined as the unique function satisfying certain equations. By observing these equations we see that they are exactly the equations used in specifying the notion of a homomorphism from $\mathcal{E}$ to $\mathcal{A}$. So we see that $\mathcal{E}$ has the special property that for every structure $\mathcal{A}$ there is a unique arrow $\mathcal{E} \to \mathcal{A}$ in the category of structures; i.e. $\mathcal{E}$ is an initial object of the category of structures, as given by the following definition.

Definition: 2.3 *An object X in a category is an* **initial object** *if, for every object Y there is a unique arrow $X \to Y$.*

In the category of sets the empty set, $\emptyset$, is the unique initial object. This is because for any set Y there is a unique function $\emptyset \to Y$, the empty function whose graph of ordered pairs is the empty set. Note that a category need not have initial objects. For example the category of non-empty sets is such a category. Nor need there be only one initial object. For example in the category of Peano structures the Peano structure of natural numbers is distinct from the syntax structure, but is also easily seen to be initial. While the two structures are distinct they are abstractly the same in that one is a copy of the other in the sense that there is a one-one correspondence which gives homomorphisms both ways between the two structures; i.e. the two structures are isomorphic. This is a general fact about initial objects - when there is an initial object, any pair of them are isomorphic with a unique isomorphism. We turn to the general notion of isomorphism in a category.

2.4 Isomorphism

In abstract mathematics mathematical structures are viewed as abstractly the same structures when they are isomorphic; i.e. when there is a pair of homomorphisms between the structures that are inverses of each other. This notion of isomorphism can be defined in any category.

In a category, arrows $f : A \to B$, $g : B \to A$ form a **pair of inverse isomorphisms** if

$$g \circ f = id_A \text{ and } f \circ g = id_B.$$

Note that g, f uniquely determine each other. For example if, for a given g both $f = f_1$ and $f = f_2$ satisfy the above equations then

$$\begin{aligned} f_1 &= f_1 \circ id_A \\ &= f_1 \circ (g \circ f_2) \\ &= (f_1 \circ g) \circ f_2 \\ &= id_B \circ f_2 \\ &= f_2. \end{aligned}$$

We write g^{-1} for the unique f, call g an **isomorphism** and write $g : B \cong A$. If there is such a g then we write $B \cong A$ and call A, B **isomorphic**.

2.5 Initial Objects

In abstract mathematics it is only relevent to characterise a mathematical structure 'up to isomorphism'; i.e. by specifying conditions on structures so that

1. There is a structure satisfying the conditions.
2. Any structure isomorphic to one satisfying the conditions itself satisfies the conditions.
3. For any pair of structures satisfying the conditions the structures are isomorphic.

Moreover if the isomorphism in 3 is unique then all the better.

A fundamental theme in category theory is the use of what has been called 'universal mapping properties' to specify objects in a category 'up to unique isomorphism'. The simplest example of this idea is given by the notion of initial object that we have already defined.

Exercise 2.4 *Show that if A is isomorphic to an initial object then A is itself an initial object.*

Proposition: 2.5 *If A_1, A_2 are initial objects then there is a unique isomorphism $A_1 \cong A_2$.*

Proof: Given the initial objects A_1, A_2 there are unique arrows $A_1 \xrightarrow{f} A_2$ and $A_2 \xrightarrow{g} A_1$. Moreover these must form a pair of inverse arrows because the identity arrows, id_{A_1} and id_{A_2} are the unique arrows $A_1 \rightarrow A_1$ and $A_2 \rightarrow A_2$.

In summary the following theorem illustrates the main point of this lecture. The 'initial algebra' approach to syntax and semantics for a formal language is to choose a category, so that the syntax of the formal language can be viewed as an initial object of the category. Then any semantics should be specified as an object in the category, and once that has been done the meaning function is uniquely determined as the unique map from the syntactic universe, given as initial object, to the object representing the semantics.

Theorem: 2.6 *In each of our four examples of categories of structures the syntax structure can be characterised up to isomorphism as an initial structure; i.e. an initial object in the category of structures.*

3 Initial Algebras and Final Coalgebras

Let us suppose that we are somehow given a formal language and we want to choose a category $\mathbb{C}$ so that the syntax can be represented as an initial object I and each interpretation can be represented as an object A and the meaning function can be represented as the unique arrow $I \rightarrow A$. In many cases, such as our four examples, the category $\mathbb{C}$ is naturally chosen as the category of algebras for an endo functor, using terminology we are about to introduce. The notion of functor $F : \mathbb{C} \rightarrow \mathbb{C}'$ from a category $\mathbb{C}$ to a category $\mathbb{C}'$ is the natural notion of homomorphism, i,e. structure preserving map between categories.

3.1 Functors

If $\mathbb{C}, \mathbb{C}'$ are categories, a **functor**, F, from $\mathbb{C}$ to $\mathbb{C}'$, written $F : \mathbb{C} \rightarrow \mathbb{C}'$, consists of

1. an assignment of an object $F(A)$ of $\mathbb{C}'$ to each object A of $\mathbb{C}$,

2. an assignment of an arrow $F(f) : F(A) \rightarrow F(B)$ of $\mathbb{C}'$ to each arrow $f : A \rightarrow B$ of $\mathbb{C}$.

Note that if (g, f) is a composable pair of $\mathbb{C}$ then $(F(g), F(f))$ will be a composable pair of $\mathbb{C}$. The following conditions must hold.

F-1 If A is an object of $\mathbb{C}$ then $F(id_A) = id_{F(A)}$,

F-2 If (g, f) is a composable pair of $\mathbb{C}$ then $F(g \circ f) = F(g) \circ F(f)$.

When $\mathbb{C}'$ is the same as $\mathbb{C}$ then F is called an **endo functor** on $\mathbb{C}$.
Examples: On any category there is the identity endo functor. Also functors between categories can be composed. So categories and functors between them have all the properties of being the objects and arrows of a category. But there is a foundational problem with the 'category of all categories', analogous to the foundational problem concerning the 'set of all sets', that I will not discuss further here. Nevertheless it is a good exercise for the category theoretic novice to show that categories and functors behave like the objects and arrows of a category.

Other examples of functors are given by the forgetful functor (also called underlying set functor) that exists for any notion of mathematical structure $\mathcal{A} = (A, \cdots)$, where A is a set, the **underlying set** of $\mathcal{A}$, as in our four examples. In each case the forgetful functor $F : \mathbb{C} \rightarrow$ **Set**, where $\mathbb{C}$ is the category of structures, $\mathcal{A}$, and homomorphisms between them, is given by defining $F(\mathcal{A})$ to be the underlying set of $\mathcal{A}$, and if π is a homomorphism from structure $\mathcal{A}$ to $\mathcal{A}'$ then defining $F(\pi)$ to be π as a function between their underlying sets. Note that, strictly speaking, F should apply to the arrow which is the triple $(\mathcal{A}, \pi, \mathcal{A}')$.

3.2 Structures as algebras

Algebras for an endo functor

Given an endo functor F on a category an **algebra for** F is a pair (A, α) where A is an object of the category and $\alpha : F(A) \rightarrow A$. We may form the category $\mathbf{alg}(F)$, whose objects are the algebras for F and whose arrows are given by homomorphisms between such algebras. If (A, α) and (A', α') are algebras for F then a **homomorphism** $(A, \alpha) \rightarrow (A', \alpha')$ is a function $\pi : A \rightarrow A'$ such that

$$\pi \circ \alpha = \alpha' \circ F(\pi).$$

So the arrows of the category can be taken to be the triples $((A, \alpha), \pi, (A', \alpha'))$ or alternatively we could just use triples (α, π, α').

We show that, in each of our four examples of notions of structure, we can define an endo functor F on the category **Set** of sets, so that each structure $\mathcal{A} = (A, \ldots)$ can be viewed as an algebra (A, α) for F. In each case below we first define $F(X)$ for each set X and for sets X, Y and each function

$\pi : X \to Y$ we define $F(\pi) : F(X) \to F(Y)$. We leave it as an exercise to check that F is indeed a functor. Finally in each case we define $\alpha : F(A) \to A$ given a structure $\mathcal{A} = (A, \ldots)$. But first we need some notation for binary and indexed disjoint unions.

Some Notation: For sets X, Y we let $X + Y$ be their **binary disjoint union**, using $0, 1$ as the labels; i.e.

$$X + Y = (\{0\} \times X) \cup (\{1\} \times Y).$$

More generally, if $(X_i)_{i \in I}$ is a family of sets X_i, indexed by the set I, then $\sum_{i \in I} X_i$ is the **indexed disjoint union** of the family; i.e.

$$\sum_{i \in I} X_i = \{(i, x) \mid i \in I \text{ and } x \in X_i\}.$$

In the definitions of the functor F in the first two examples below we use 1 for the singleton set $\{0\}$.

Example 1: $F(X) = 1 + X$

$$F(\pi) : 1 + X \to 1 + Y$$

$$\begin{cases} F(\pi)(0,0) & = (0,0) \\ F(\pi)(1,x) & = (1, \pi(x)) \text{ for all } x \in X \end{cases}$$

$$\alpha : 1 + A \to A$$

$$\begin{cases} \alpha(0,0) & = a \\ \alpha(1,x) & = f(x) \text{ for all } x \in A \end{cases}$$

Example 2: $F(X) = 1 + (K \times X)$

$$F(\pi) : 1 + (K \times X) \to 1 + (K \times Y)$$

$$\begin{cases} F(\pi)(0,0) & = (0,0) \\ F(\pi)(1,(k,x)) & = (1,(k,\pi(x))) \text{ for all } k \in K, x \in X \end{cases}$$

$$\alpha : 1 + (K \times A) \to A$$

$$\begin{cases} \alpha(0,0) & = a \\ \alpha(1,(k,x)) & = f(k,x) \text{ for all } k \in K, x \in A \end{cases}$$

Example 3: $F(X) = K + (X \times X)$

$$F(\pi) : K + (X \times X) \to K + (Y \times Y)$$

$$\begin{cases} F(\pi)(0,k) & = (0,k) \text{ for all } k \in K \\ F(\pi)(1,(x_1,x_2)) & = (1,(\pi(x_1),\pi(x_2))) \text{ for all } x_1, x_2 \in X \end{cases}$$

$\alpha : K + ((A \times A) \rightarrow A$

$$\begin{cases} \alpha(0, k) & = a(k) \text{ for all } k \in K \\ \alpha(1, (x_1, x_2)) & = f(x_1, x_2) \text{ for all } x_1, x_2 \in A \end{cases}$$

Example 4: $F(X) = \sum_{\sigma \in \Sigma} X^{n_\sigma}$

$F(\pi) : \sum_{\sigma \in \Sigma} X^{n_\sigma} \rightarrow \sum_{\sigma \in \Sigma} Y^{n_\sigma}$

$$\begin{aligned} F(\pi)(\sigma, (x_1, \ldots, x_{n_\sigma})) &= (\sigma, (\pi(x_1), \ldots, \pi(x_{n_\sigma})) \\ &\quad \text{for all } x_1, \ldots, x_{n_\sigma} \in X. \end{aligned}$$

$\alpha : \sum_{\sigma \in \Sigma} A^{n_\sigma} \rightarrow A$

$$\begin{aligned} \alpha(\sigma, (x_1, \ldots, x_{n_\sigma})) &= \sigma^{\mathcal{A}}(x_1, \ldots, x_{n_\sigma})) \\ &\quad \text{for all } x_1, \ldots, x_{n_\sigma} \in A. \end{aligned}$$

Exercise 3.1 *Show that in each case above F is indeed a functor. Also show that the assignment of an algebra (A, α) to each structure $\mathcal{A} = (A, \ldots)$ determines a functor from the category of structures to the category of algebras that is an isomorphism of categories; i.e. that is a one-one correspondence on both objects and arrows.*

3.3 Initial Algebras

Under what conditions will a category of algebras for an endo functor have an initial algebra? We give an answer to this question for an endo functor on the category of sets.

Bounded Standard Endo Functors

An endo functor F on the category **Set** is **standard** if, for all sets A, B, such that A is a subset of B, the set $F(A)$ is a subset of $F(B)$ and

$$F(\iota_{A,B}) = \iota_{F(A),F(B)},$$

where $\iota_{A,B} : A \rightarrow B$ is the **inclusion** function from A to B; i.e. the unique function from A to B such that $\iota_{A,B}(x) = x$ for all $x \in A$.

A standard functor F is **bounded standard**, with bound κ, if κ is a regular cardinal number such that, for any set A if $x \in F(A)$ then there is a subset A' of A of cardinality $< \kappa$, such that $x \in F(A')$. If κ is $\aleph_0$ then we call F **finitary standard**.

Exercise 3.2 *Show that the four examples of functors given in 3.2 are all finitary standard.*

The Initial Algebra Theorem

We will obtain initial algebras via the following fixed point theorem.

Theorem: 3.3 *Let F be bounded standard. Then, as an operator on sets, F has a unique least fixed point I; i.e. I is a set such that $F(I) = I$ and if $F(X) = X$ then $I \subseteq X$.*[1]

Sketch Proof in the finitary case: Let $I = I^0 \cup I^1 \cup I^2 \cup \cdots$ where the I^n are defined by primitive recursion as the unique sets such that $I^0 = \emptyset$ and, for $n \in \mathbb{N}$, $I^{n+1} = F(I^n)$. By mathematical induction $I^0 \subseteq I^1 \subseteq I^2 \subseteq \cdots$. To see that $F(I) \subseteq I$ observe that, because F is finitary, if $x \in F(I)$ then $x \in F(I^n) = I^{n+1} \subseteq I$ for some $n \in \mathbb{N}$. Now if X is a set such that $F(X) \subseteq X$ then an easy mathematical induction shows that $I^n \subseteq X$ for all $n \in \mathbb{N}$, so that we get $I \subseteq X$. Finally, we can apply this last fact to the case when $X = F(I)$ to get that $I \subseteq F(I)$ so that I is seen to be the least fixed point of F.

Exercise 3.4 *Fill in the details in the above sketch proof.*

The above proof can be generalised to give a proof of the full theorem. When the bound κ is larger than $\aleph_0$ the finite iterations I^n need to be extended to transfinite iterations I^λ for all ordinals λ below κ.

We are now ready to give an answer to the question raised at the start of this section.

Theorem: 3.5 (Initial Algebra Theorem)
Every bounded standard functor has an initial algebra. In fact, if I is the least fixed point of F then (I, id_I) is an initial algebra for F.

Sketch Proof in the finitary case: We use the definition of I in terms of the finite iterations of F used in the previous sketch proof. Given an algebra (A, α) we define, by primitive recursion, functions $\pi_n : I^n \to A$ for $n \in \mathbb{N}$. Let π_0 be the empty function $\emptyset \to A$ and, for $n \in \mathbb{N}$, let $\pi_{n+1} = \alpha \circ F(\pi_n)$. For $n \in \mathbb{N}$ let $i_n : I^n \to I^{n+1}$ be the inclusion function. Then by mathematical induction and the fact that F is standard we get that $\pi_n = \pi_{n+1} \circ i_n$ for all $n \in \mathbb{N}$. It follows that we can now define $\pi : I \to A$. For $x \in I$ let

$$\pi(x) = \pi_n(x)$$

for any $n \in \mathbb{N}$ large enough so that $x \in I^n$. Now let $i_{n\infty}$ be the inclusion function $I^n \to I$ for each $n \in \mathbb{N}$. Then, for $n \in \mathbb{N}$, $\pi_n = \pi \circ i_{n\infty}$ and, as F is standard, $F(i_{n\infty}) = i_{(n+1)\infty}$ so that

$$\begin{aligned} \pi \circ i_{n+1} &= \pi_{n+1} \\ &= \alpha \circ F(\pi_n) \\ &= \alpha \circ F(\pi) \circ F(i_{n\infty}) \\ &= \alpha \circ F(\pi) \circ i_{(n+1)\infty} \end{aligned}$$

[1] More generally, if $F(X) \subseteq X$ then $I \subseteq X$.

It is now easy to see that $\pi = \alpha \circ F(\pi)$ so that π is a coalgebra homomorphism $(I, id_I) \rightarrow (A, \alpha)$. To show that π is the unique such homomorphism let π' be any such homomorphism. By mathematical induction on $n \in \mathbb{N}$ we get that $\pi' \circ i_{n\infty} = \pi_n$ and hence that $\pi' = \pi$.

Exercise 3.6 *Fill in the details in the above sketch proof.*

The proof of the full result again involves transfinite iterations.

3.4 Final Coalgebras

Duality

If $\mathbb{C}$ is a category then we may form the **dual** or **opposite** category $\mathbb{C}^{op}$ by interchanging the roles of the domain and codomain operations. So, if $f : A \rightarrow B$ in $\mathbb{C}$ then $f : B \rightarrow A$ in $\mathbb{C}^{op}$ and visa versa. Note that $(\mathbb{C}^{op})^{op} = \mathbb{C}$. Any notion or property of categories will have a dual notion or property by interpreting it in the dual category. Some notions are self dual; e.g. identity arrows and the notion of isomorphism. Note that the dual of a composable pair (g, f) is the pair (f, g).

Given a functor $F : \mathbb{C}_1 \rightarrow \mathbb{C}_2$ we may define its dual functor $F^{op} : \mathbb{C}_1^{op} \rightarrow \mathbb{C}_2^{op}$, which acts just like F.

Final Objects

An object X in a category is a **final object** if, for every object Y there is a unique arrow $Y \rightarrow X$.

Note that this notion is dual to the notion of an initial object; i.e. an object is final in a category iff it is initial in the dual category. Hence we get the following dual version of an earlier result.

Proposition: 3.7 *If A_1, A_2 are final objects then there is a unique isomorphism $A_1 \cong A_2$.*

Example: In the category of sets the final objects are the singleton sets.

Coalgebras

Given an endo functor F on a category, a **coalgebra for** F is defined to be an algebra for F^{op}; i.e. it is a pair (A, α), where A is an object of the category and $\alpha : F(A) \rightarrow A$ in $\mathbb{C}^{op}$, which means that $\alpha : A \rightarrow F(A)$ in $\mathbb{C}$. We can define homomorphisms between coalgebras in the obvious way, so that the category, $\mathbf{coalg}(F)$, of coalgebras for F can be defined to be the category $\mathbf{alg}(F^{op})$.

The Final Coalgebra Theorem

The following result is dual, in some sense, to the Initial Algebra Theorem. But the proof of the result is NOT at all dual.

Theorem: 3.8 (Final Coalgebra Theorem)
Every bounded standard functor has a final coalgebra.

The final coalgebra theorem will not be proved here, even in the finitary case. Ideas for a proof may be found in [1], [2] and [4], the first two references taking a somewhat set-theoretical approach while the last reference uses more advanced category theory ideas than are covered here.

Instead here we will give final coalgebras for the four example finitary functors that we have been using. In general, while the elements of an initial algebra can be viewed as certain kinds of well-founded trees, the particular kind depending on the functor, the elements of the final coalgebra can be viewed as the same kind of trees, but with the well-foundedness condition dropped.

Full algebras and full coalgebras

An algebra (A, α), for an endo functor F is **full** if $\alpha : F(A) \to A$ is an isomorphism. Dually, a coalgebra (A, β) is **full** if $\beta : A \to F(A)$ is an isomorphism. Note that by taking inverses we get a one-one correspondence between full algebras, with a given underlying set, and full coalgebras with the same underlying set. Observe that in each of the syntax structures of 1.5 the associated algebras are all full. More generally you should verify the following result.

Exercise 3.9

1. *Any initial algebra for an endo functor is full.*
2. *Any full coalgebra for an endo functor is full.*

Note that these form a dual pair of results, so that each of them follows from the other by duality. The idea for the proof of 1 is to observe that if (A, α) is an algebra then so is $(F(A), F(\alpha))$ and if (A, α) is initial then there is a unique arrow $(A, \alpha) \to (F(A), F(\alpha))$ which determines the inverse arrow $A \to F(A)$ to the arrow $\alpha : F(A) \to A$.

3.5 Examples of final coalgebras

Each example below of a final coalgebra will be obtained as the full coalgebra associated with a full algebra that comes from a structure we have already described in 1.4.

Example 1:

We start with the example, $(\mathbb{N}^\infty, 0, s^\infty)$, of a Peano structure given in 1.4. The functor F for Peano structures associates the set $1 + X$ with each set X. The algebra $(\mathbb{N}^\infty, \gamma)$ for this functor that is associated with the Peano structure has $\gamma : (1 + \mathbb{N}^\infty) \to \mathbb{N}^\infty$ given by

$$\begin{cases} \gamma(0,0) & = 0 \\ \gamma(1,n) & = n+1 \text{ for all } n \in \mathbb{N} \\ \gamma(1,\infty) & = \infty \end{cases}$$

This algebra is clearly full, so that if $\eta = \gamma^{-1}$ then we get a full coalgebra $(\mathbb{N}^\infty, \eta)$.

Theorem: 3.10 *The full coalgebra $(\mathbb{N}^\infty, \eta)$ is a final coalgebra.*

To see this let (A, α) be any coalgebra, so that $\alpha : A \to 1 + A$. Then the required unique homomorphism $\pi : (A, \alpha) \to (\mathbb{N}^\infty, \eta)$ can be explicitly defined as follows. If $a \in A$ then there is a uniquely determined finite or infinite sequence $a_0, \ldots$ of elements of A, starting with $a_0 = a$, such that for each item a_n in the sequence either $\alpha(a_n) = (0,0)$ and a_n ends the sequence or else there is a next item a_{n+1} and $\alpha(a_n) = (1, a_{n+1})$. There are two possibilities. Either the first case eventually arises and the sequence is finite, ending in a_n, or the first case never arises and the sequence is infinite. In the first case we let $\pi(a) = n$ and in the second case we let $\pi(a) = \infty$.

Exercise 3.11 *Check that π is indeed the unique homomorphism.*

Example 2:

We now consider the finitary standard endo functor '$X \mapsto 1 + (K \times X)$', which is the one used in our second example of the formal language for list structures. In 1.4 we introduced the List structure $(K^\infty, \epsilon, cons^\infty)$ of finite and infinite strings. The associated algebra (K^∞, γ) has $\gamma : 1 + (K \times K^\infty) \to K^\infty$ given by

$$\begin{cases} \gamma(0,0) & = \epsilon \\ \gamma(1,(k,\rho)) & = cons^\infty(k,\rho) \text{ for all } k \in K, \rho \in K^\infty \end{cases}$$

Again this algebra is easily seen to be full so that we get a full coalgebra (K^∞, η) by letting $\eta = \gamma^{-1}$.

Theorem: 3.12 *The full coalgebra (K^∞, η) is a final coalgebra.*

To see this let (A, α) be any coalgebra, so that $\alpha : A \to 1 + (K \times A)$. Then the required unique homomorphism $\pi : (A, \alpha) \to (K^\infty, \eta)$ can be explicitly defined as follows. If $a \in A$ then there is a uniquely determined finite or infinite sequence $a_0, k_1, a_1, \ldots$ of items that are alternately elements of A

and elements of K, starting with $a_0 = a$, such that for each item a_n either $\alpha(a_n) = (0, 0)$ and a_n ends the sequence, or else there are two following items k_{n+1}, a_{n+1} and $\alpha(a_n) = (1, (k_{n+1}, a_{n+1}))$. There are two possibilities. Either the first case eventually arises and the sequence is finite, ending in a_n, or the first case never arises and the sequence is infinite. In the first case we let $\pi(a) = k_1 \cdots k_n \in K^*$ and in the second case we let $\pi(a) = k_1 \cdots \in K^\omega$.

Exercise 3.13 *Check that π is indeed the unique homomorphism.*

Example 3:

In 1.4 we defined the set $\mathbb{S}^\infty$ of S-trees which naturally formed a Lisp structure. This determines an algebra $(\mathbb{S}^\infty, \gamma)$ for the functor F for Lisp structures. The function $\gamma : K + (\mathbb{S}^\infty \times \mathbb{S}^\infty) \to \mathbb{S}^\infty$ is given by

$$\begin{cases} \gamma(0, k) & = atom(k) \text{ for all } k \in K \\ \gamma(1, (M_1, M_2)) & = M_1 \cdot M_2 \text{ for all } M_1, M_2 \in \mathbb{S}^\infty \end{cases}$$

By exercise 1.2 this algebra is full and so we can define a full coalgebra $(\mathbb{S}^\infty, \eta)$, where $\eta = \gamma^{-1}$.

Theorem: 3.14 *The full coalgebra $(\mathbb{S}^\infty, \eta)$ is a final coalgebra.*

Proof: Let (A, α) be a coalgebra. So $\alpha : A \to K + (A \times A)$. Given $a \in A$ we will describe a branching procedure $\mathsf{proc}(a)$ that we will be able to use to determine an element $M_a \in \mathbb{S}^\infty$. The assigment of M_a to each $a \in A$ will give us the unique arrow $(A, \alpha) \to (\mathbb{S}^\infty, \eta)$ that we need. The procedure $\mathsf{proc}(a)$ will associate with each of certain strings $\rho \in \{1, 2\}^*$ an element $a_\rho \in A$ and then will perform a step $step(\rho)$ which will either halt the procedure on that branch with a value $k_\rho \in K$ or else will cause the branch to split into two sub-branches that determine elements $a_{\rho 1}, a_{\rho 2} \in A$ associated with the strings $\rho 1, \rho 2 \in \{1, 2\}^*$.

At the start of $\mathsf{proc}(a)$ we put $a_\epsilon = a$. If $\rho \in \{1, 2\}^*$ so that $a_\rho \in A$ has been determined then $step(\rho)$ is performed as follows. Consider $\alpha(a_\rho) \in K + (A \times A)$. There are two cases

Case 1: $\alpha(a_\rho) = (0, k)$ for some $k \in K$. In this case the procedure halts with value $k_\rho = k$.

Case 2: $\alpha(a_\rho) = (1, (a^1, a^2))$ for some $a^1, a^2 \in A$. This time the procedure splits with elements $a_{\rho 1} = a^1 \in A$ and $a_{\rho 2} = a^2 \in A$ associated with the strings $\rho 1, \rho 2$ so that further steps $step(\rho 1)$, $step(\rho 2)$ can then be performed.

Having described the procedure $\mathsf{proc}(a)$ we can now define $M_a \in \mathbb{S}^\infty$. We let $M_a = (|M_a|, l_{M_a})$ where

$$\begin{array}{lll} |M_a| & = \{\rho \mid a_\rho \text{ is determined in } \mathsf{proc}(a)\} & \\ l_{M_a}(\rho) & = \begin{cases} k_\rho & \text{if } step(\rho) \text{ is a halt step} \\ \cdot & \text{otherwise} \end{cases} & \text{if } \rho \in |M_a|. \end{array}$$

Exercise 3.15 *Show that, for each $a \in A$, M_a is an S-tree. Let $\pi : A \to \mathbb{S}^\infty$ be given by $\pi(a) = M_a$ for $a \in A$. Show that π is the unique arrow $\pi : (A, \alpha) \to (\mathbb{S}^\infty, \eta)$.*

Example 4:

We now turn to the description of the final coalgebra of Σ-trees. This will be a smooth generalisation of the previous example. This time we start with the Σ-structure $(\mathbb{T}^\infty\Sigma, \ldots)$ described in 1.4. The functor for Σ-structures assigns to each set X the set $\sum_{\sigma\in\Sigma} X^{n_\sigma}$. So the algebra for this functor, that is determined by the Σ-structure, is $(\mathbb{T}^\infty\Sigma, \gamma)$ where $\gamma : \sum_{\sigma\in\Sigma} \mathbb{T}^\infty\Sigma^{n_\sigma} \to \mathbb{T}^\infty\Sigma$ is given by

$$\gamma(\sigma, (M_1, \ldots, M_{n_\sigma})) = \sigma(M_1, \ldots, M_{n_\sigma})$$

for $\sigma \in \Sigma$, $M_1, \ldots, M_{n_\sigma} \in \mathbb{T}^\infty\Sigma$. By exercise 1.2 this algebra is full so that it determines the full coalgebra $(\mathbb{T}^\infty\Sigma, \eta)$ where $\eta = \gamma^{-1}$. By a straightforward generalisation of the details of the proof of theorem 3.14 we get

Theorem: 3.16 *The full coalgebra $(\mathbb{T}^\infty\Sigma, \eta)$ is a final coalgebra.*

Exercise 3.17 *Work out the details of a proof of this theorem.*

4 Final Universes of Processes

The aim of this lecture is to illustrate the general theme that often a paradigm of computation, i.e. a style of computational model, can be naturally represented in terms of the category of coalgebras for a suitably chosen endo functor Then a final coalgebra in the category can play the role of a semantic universe for the paradigm. Elements of the final coalgebra can be viewed as the abstract processes associated with the paradigm.

4.1 The non-interactive, deterministic paradigm

Today we are aware of many paradigms of computation, giving rise to a great variety of computational models. But in the early days of computing the main focus of attention was on the fundamental paradigm exemplified on paper by Turing machines and in practice by computers based on the von Neumann idea. Abstracting away from many details including the necessary effectiveness of computation steps we may say that this paradigm is concerned with computations that are **non-interactive** and **deterministic**. According to this paradigm a computation consists of a finite or infinite sequence $s_0 \to s_1 \to s_2 \to \cdots$ of consecutive steps going from one computation state s_n to the next s_{n+1}. At the start of the computation the state s_0 is determined by some input data *in*, taken from a set *In* of possible items of input data, each subsequent state s_{n+1} being determined uniquely by its previous state s_n, and if the computation terminates in a state s_N then that state determines some output data *out* belonging to a set Out of possible items of output data. So the whole computation, with its output when terminating, is uniqely determined by the input. Moreover there is no interaction with the environment of the computation other than at the start and on termination. We may represent this computational model by a pair of functions

$$\begin{cases} input & : In \to State \\ step & : State \to Out + State \end{cases}$$

where $State$ is the set of possible computation states, the function $input$ gives the starting state $s_0 = input(in)$ determined by $in \in In$ and given $s_n \in State$ either $step(s_n) = (0, out)$ when the computation terminates at s_n with output out or else $step(s_n) = (1, s_{n+1})$ when there is a computation step $s_n \longrightarrow s_{n+1}$.

The important point is to note is that we have a coalgebra $(State, step)$ for the finitary standard endo functor $X \mapsto (Out + X)$. We are interested in a final coalgebra for this endo functor.

Exercise 4.1 *Show that* $(\mathbb{N}^\infty(Out), \eta)$ *is a final coalgebra for the above endo functor, where* $\mathbb{N}^\infty(Out) = (\mathbb{N} \times Out) \cup \{\infty\}$ *and* $\eta : \mathbb{N}^\infty(Out) \to (Out + \mathbb{N}^\infty(Out))$ *is given by*

$$\begin{cases} \eta(\infty) & = (1, \infty) \\ \eta(0, out) & = (0, out) \\ \eta(n+1, out) & = (1, (n, out)) \text{ for } n \in \mathbb{N} \end{cases}$$

By this exercise we know that there is a unique homomorphism

$$[[\,]] : (State, step) \to (\mathbb{N}^\infty(Out), \eta).$$

Given $in \in In$ we can view $[[input(in)]]$ as the abstract process that represents the behaviour of the computation determined by in when we abstract away from the internal details of computational states. This abstract process is either (n, out) for some $n \in \mathbb{N}$ indicating that the computation terminates after n steps yielding out or else is ∞ indicating that the computation is non-terminating.

Abacus Programs

We illustrate the non-interactive, deterministic paradigm by describing a simple universal computational model that follows the paradigm. An **abacus program** is defined to be a finite sequence of commands. The forms of command will be described later. The sequence of commands of a program p will be written $c_1; c_2; \ldots; c_n$ and such a program is intended to be executed by executing each command c_i in turn starting with c_1. The empty program will be written $\Box$ and we will write $c; p$ for the program $c; c_1; c_2; \ldots; c_n$ and more generally $p_1; p_2$ for the program consisting of the commands of p_1 followed by the commands of p_2. Note that every program is uniquely either of the form $\Box$ or $c;p$, so that the set of programs form an initial List structure over the set of commands.

The commands have one of the forms $r_k+, r_k-, \omega_k(p)$ where k is any natural number and p is any program. We assume available **registers** $r_0, r_1, \ldots$, each capable of storing a natural number. The commands r_k+ and r_k- are executed by incrementing the register r_k and decrementing, when possible, the register r_k and might be written more familiarly as

$$r_k := r_k + 1 \quad \text{and} \quad \mathbf{if}\ r_k > 0\ \mathbf{then}\ r_k := r_k - 1.$$

The command $\omega_k(p)$ is a while loop command with body p and loop condition $r_k > 0$ and might be written more familiarly as

$$\mathbf{while}\ r_k > 0\ \mathbf{do}\ p.$$

In summary the abstract syntax for abacus programs can be given using the following syntax equations.

$$c ::= \ r_k+ \ \mid \ r_k- \ \mid \ \omega_k(p),$$

$$p ::= \ \Box \ \mid \ c;p.$$

We have indicated an informal semantics for the execution of commands and programs. We now want to represent the informal semantics as a precise operational semantics. We will represent a computation state as a pair $s = (m, p)$ where m is the state of the memory, called just memory for short, and p is the program still to be executed. A memory is defined to be an infinite sequence $m = m_0 m_1 \cdots$ of natural numbers, m_k representing the content of the register r_k. The terminal states are those of the form $(m, \Box)$ and each non-terminal state $(m, c;p)$ determines the computation step

$$(m, c;p) \rightarrow (m', p')$$

where m', p' are determined as follows depending on the form of the command c.

c	m'	p'
r_k+	$incr_k(m)$	p
r_k-	$decr_k(m)$	p
$\omega_k(p_0)$	m	$\begin{cases} p_0;c;p & \text{if } m_k > 0 \\ p & \text{if } m_k = 0 \end{cases}$

Here, for each k, $incr_k(m)$ and $decr_k(m)$ are memories m^+ and m^- that are just like m except that $m_k^+ = m_k + 1$ and if $m_k > 0$ then $m_k^- = m_k - 1$.

We can now express this operational semantics in terms of functions

$$\begin{array}{ll} input & : Prog \times Mem \rightarrow State \\ step & : State \rightarrow Mem + State \end{array}$$

where $Prog, Mem, State$ are the sets of programs, memories and states respectively. If $m \in Mem$, $p \in Prog$, c is a command and (m', p') is as specified above then let

$$\begin{array}{ll} input(p,m) & = (m,p), \\ step(m,\Box) & = (0,m), \\ step(m,c;p) & = (1,(m',p')). \end{array}$$

So our operational semantics for abacus programs gives rise to the coalgebra $(State, step)$ and hence to the unique homomorphism $[[\ ]] : (State, step) \to (\mathbb{N}^\infty(Mem), \eta)$ into our universe of processes $\mathbb{N}^\infty(Mem)$.

In the following section we show how this operational semantics can be rephrased in denotational terms; i.e. we represent the abstract syntax as an initial object in a suitably chosen category and rephrase the above function into $\mathbb{N}^\infty(Mem)$ as the unique arrow into a suitably chosen object of the category.

Interlude: A denotational semantics for abacus programs

We have formulated the abstract syntax for abacus programs using two syntactic categories, that of programs and also that of commands. But a program is abstractly just a list of commands, so is either empty or else is one of the three forms of command followed by a program; i.e a program has one of the forms $\Box$, $r_k+;p$, $r_k-;p$ or $\omega_k(p_0);p$, where p_0, p are programs. So, abstractly, we can just as well view programs as the expressions in the syntactic Σ-structure, for a suitable signature Σ. We will use the symbol $\Box$ with arity 0 for the program $\Box$ and, for each $k \in \mathbb{N}$, symbols $k+$ and $k-$ with arity 1 for the programs having the forms $r_k+;p$ or $r_k-;p$ and symbols ωk with arity 2 for the programs $\omega_k(p_0);p$. With this choice of signature we have a representation of the abstract syntax as an initial object $\mathcal{P} = (Prog, \ldots)$ in the category of Σ-structures, where, for example $(\omega k)^{\mathcal{P}} p_0 p_1 = \omega_k(p_0);p_1$.

We now want to define a Σ-structure $(Den, \ldots)$, where Den is the set $Mem \to \mathbb{N}^\infty(Mem)$ of all functions from Mem to $\mathbb{N}^\infty(Mem)$, in such a way that $\langle\langle\ \rangle\rangle : Prog \to Den$ is a Σ-structure homomorphism from $(Prog, \ldots)$ to $(Den, \ldots)$, where

$$\langle\langle p \rangle\rangle(m) = [[(m,p)]]$$

for $p \in Prog$ and $m \in Mem$. We outline this in a sequence of definitions and exercises.

Some Definitions

- For each $k \in \mathbb{N}$ let $(\ )^{+k} : \mathbb{N}^\infty(Mem) \to \mathbb{N}^\infty(Mem)$ be given by

$$\begin{cases} (n,m)^{+k} & = (n+k, m) \text{ for } (n,m) \in \mathbb{N} \times Mem \\ \infty^{+k} & = \infty \end{cases}$$

- Given $d \in Den$ let $d^{+} : \mathbb{N}^{\infty}(Mem) \to \mathbb{N}^{\infty}(Mem)$ be given by

$$\begin{cases} d^{+}(n,m) & = d(m)^{+n} \text{ for } (n,m) \in \mathbb{N} \times Mem \\ d^{+}(\infty) & = \infty \end{cases}$$

- Given $k \in \mathbb{N}$ and $d_0, d_1 \in Den$ let $loop_k(d_0, d_1)$ be the unique $d \in Den$ such that for all $m \in Mem$

$$(*) \qquad d(m) = \begin{cases} d^{+}(d_0(m)) & \text{if } m_k > 0 \\ d_1(m) & \text{if } m_k = 0 \end{cases}$$

Exercise 4.2 *Show that $loop_k$, above, is well defined; i.e. show that there is indeed a unique $d \in Den$ satisfying the equation $(*)$ for all $m \in Mem$.*

Exercise 4.3 *Show that for all $k \in \mathbb{N}$, $p_0, p \in Prog$ and $m \in Mem$*

$$\begin{cases} \langle\!\langle \Box \rangle\!\rangle(m) & = (0, m), \\ \langle\!\langle r_k+; p \rangle\!\rangle(m) & = \langle\!\langle p \rangle\!\rangle^{+}(1, incr_k(m)), \\ \langle\!\langle r_k-; p \rangle\!\rangle(m) & = \langle\!\langle p \rangle\!\rangle^{+}(1, decr_k(m)), \\ \langle\!\langle \omega(p_0); p \rangle\!\rangle(m) & = loop_k(\langle\!\langle p_0 \rangle\!\rangle, \langle\!\langle p \rangle\!\rangle)(m). \end{cases}$$

Hence determine a Σ-structure $(Den, \ldots)$ so that $\langle\!\langle\ \rangle\!\rangle$ is the unique homomorphism $(Prog, \ldots) \to (Den, \ldots)$.

Note that our denotational semantics keeps track, in the denotation, of the number of computation steps, when the computation is terminating. The more standard kind of denotational semantics does not do this. We can recapture the standard kind of semantics with set $Den' = Mem \cup \{\infty\}$ of denotations by letting $\langle\!\langle\ \rangle\!\rangle' : Prog \to Den'$ be given by

$$\langle\!\langle p \rangle\!\rangle'(m) = \begin{cases} m' & \text{if } \langle\!\langle p \rangle\!\rangle(m) = (n, m'), \\ \infty & \text{if } \langle\!\langle p \rangle\!\rangle(m) = \infty, \end{cases}$$

for $p \in Prog$ and $m \in Mem$.

Exercise 4.4 *Determine a Σ-structure $(Den', \ldots)$ so that $\langle\!\langle\ \rangle\!\rangle'$ is the unique homomorphism $(Prog, \ldots) \to (Den', \ldots)$.*

4.2 Interactive, deterministic paradigms

In non-interactive paradigms of computation, after receiving the initial input a computation does not communicate with its environment unless and until the computation terminates to yield an output value. It is possible to envisage various possible kinds of observable interaction at the intermediate states of a computation. We will consider two examples. In the first there are input and output interactions at each computation step. In the second we abstract away from the distinction between inputs and outputs and just focus on atomic actions to represent the observable interactions. In both examples the state after a computation step is uniquely determined by the state of the computation before the step and the observable interaction information. It is in this sense that we are still concerned with deterministic paradigms.

First Example: In this example a computation state a is either terminal or else an input item $in \in In$ is requested and an output item $out \in Out$ is determined resulting in a computation step $a \xrightarrow{\mu} a'$ to a state a', where $\mu = (in, out)$. The appropriate functor for this paradigm is the bounded standard functor F where, for each set X

$$F(X) = 1 + (In \to (Out \times X)).$$

So each element of $F(X)$ either has the form $(0,0)$ or else the form $(1, f)$ where $f : In \to (Out \times X))$.

Exercise 4.5 *Show how to define F on functions between sets.*

If (A, α) is a coalgebra for this functor then $\alpha : A \to (In \to (Out \times A))$ and if $a \in A$ then a is a terminal state if $\alpha(a) = (0,0)$ and if a is not terminal, with $\alpha(a) = (1, f)$, then for each $in \in In$, if $f(in) = (out, a')$ and $\mu = (in, out)$ then there is a computation step $a \xrightarrow{\mu} a'$. In general, starting from a state $a_0 \in A$, there will be computations of the form

$$a_0 \xrightarrow{\mu_1} a_1 \xrightarrow{\mu_2} a_2 \cdots$$

with $\mu_1 = (in_1, out_1)$, $\mu_2 = (in_2, out_2)$, ..., determined by the input items $in_1, in_2, \ldots$ provided by the environment. Such a computation will either eventually reach a terminal state or else will be infinite.

We do not pursue this example in any more detail here as it can be covered in a certain sense by our second example. See Exercise 4.8.

Second Example: We assume given a set Act of **atomic actions**. These atomic actions are intended to represent observable interactions between a process and its environment. They are atomic in the sense that they represent what can happen during what we treat as a single computation step. The previous example will be covered by taking $Act = In \times Out$. There will be no specifically terminal states, as a computation can stop in any state. But in a given state a only certain atomic actions $\mu \in Act$ will be allowed and for each these there will be a uniquely determined step $a \xrightarrow{\mu} a'$ to a state a'.

An appropriate endo functor for this paradigm is the bounded standard functor F, where for each set X

$$F(X) = Act \to (1 + X).$$

Exercise 4.6 *Show how to define F on functions between sets.*

If $\alpha : A \to (Act \to (1 + A))$ is a coalgebra for this functor then $a \in A$ allows $\mu \in Act$ if $\alpha(a)(\mu) = (1, a')$ for some $a' \in A$ and then there is a computation step $a \xrightarrow{\mu} a'$. In general a computation will have the form

$$a_0 \xrightarrow{\mu_1} a_1 \xrightarrow{\mu_2} a_2 \cdots.$$

We now describe a final coalgebra for the above endo functor. The elements of the coalgebra will be trace sets. A **trace set**, relative to the set Act, is a set $X \subseteq Act^*$ of strings of atomic actions that is non-empty and prefix closed. The latter property means that for any string $\sigma \in Act^*$ and any $\mu \in Act$ if $\sigma\mu \in X$ then $\sigma \in X$. Note that any trace set must contain the empty string. Let TS be the set of all trace sets. To obtain the coalgebra (TS, η) we define $\eta : TS \to (Act \to (1 + TS))$. If $X \in TS$ and $\mu \in Act$ let

$$\eta(X)(\mu) = \begin{cases} (0,0) & \text{if } \mu \notin X \\ (1, X_\mu) & \text{if } \mu \in X. \end{cases}$$

Here $X_\mu = \{\sigma \in Act^* \mid \mu\sigma \in X\}$. Note that whenever $\mu \in X$ then X_μ is a trace set.

Exercise 4.7 *Show that (TS, η) is a final coalgebra.*

Exercise 4.8 *In this exercise we assume that $Act = In \times Out$. Show that (TS_0, η_0) is a final coalgebra for the functor of the first example, that maps each set X to the set $1 + (In \to (Out \times X))$. We define TS_0 to be the set of those trace sets X such that for every $\sigma \in X$ either*

(i) *$\sigma\mu \notin X$ for all $\mu \in Act$*

or

(ii) *for every $in \in In$ there is a unique $out \in Out$ such that $\sigma\mu \in X$ where $\mu = (in, out)$.*

Let η_0 be the function $TS_0 \to 1 + (In \to (Out \times TS_0))$, where for $X \in TS_0$

$$\eta_0(X) = \begin{cases} (0,0) & \text{if } X = \{\epsilon\} \\ (1, f_X) & \text{if } X \neq \{\epsilon\}. \end{cases}$$

Here ϵ is the empty string and if $X \neq \{\epsilon\}$ then f_X is the function $In \to (Out \times TS_0)$ where for each $in \in In$

$$f_X(in) = (out, X_\mu)$$

where out is the unique element of Out such that $(in, out) \in X$ and $\mu = (in, out)$.

4.3 Interactive, non-deterministic paradigms

Labelled Transition Systems

In this section we will keep to the abstract treatment of interaction using a set Act of atomic actions, but now allowing non-deterministic behaviour. This means that, given $\mu \in Act$, from a state a there may now be zero, one or more computation steps $a \xrightarrow{\mu} a'$. So we are led to the notion of a **Labelled Transition System**, abbreviated LTS. An LTS $(A, \xrightarrow{\mu})_{\mu \in Act}$ consists of a set A together with binary relations $\xrightarrow{\mu}$ on A, one for each $\mu \in Act$. The notion of an LTS has been fundamental in the mathematical study of concurrency. It plays an important role in connection with the operational semantics of Robin Milner's CCS (the Calculus of Communicating Systems). The calculus CCS has been central in the process algebra approach to concurrent processes. Our aim here is to briefly review the final coalgebra approach to CCS agents, up to strong bisimulation.

The first point is to observe that an LTS an be viewed as a coalgebra for the standard endo functor F where, for any set X,

$$F(X) = pow(Act \times X)$$

and for any function $f : X \to Y$ thwe function $F(f) : F(X) \to F(Y)$ is given by

$$F(f)(s) = \{(\mu, f(x)) \mid (\mu, x) \in s\}$$

for all sets $s \subseteq Act \times X$. Here, for any set Y, $pow(Y)$ is the set of all subsets of Y.

Exercise 4.9 *Show that F is indeed a standard functor.*

Each LTS $(A, \xrightarrow{\mu})_{\mu \in Act}$ determines a coalgebra (A, α) where the function $\alpha : A \to pow(Act \times A)$ is given by

$$\alpha(a) = \{(\mu.a') \mid a \xrightarrow{\mu} a'\}$$

for each $a \in A$. This gives a one-one correspondence between LTSs and coalgebras, as any coalgebra (A, α) clearly corresponds to the LTS $(A, \xrightarrow{\mu})_{\mu \in Act}$ where

$$a \xrightarrow{\mu} a' \;\Leftrightarrow\; (\mu, a') \in \alpha(a).$$

Unfortunately the standard endo functor F is not bounded, so that we cannot use the Final Coalgebra Theorem to get a final coalgebra, and in fact there is no final coalgebra because of a problem of size. The construction of a 'final coalgebra' would give a proper class of elements. The approach taken in [2, 1, 3] is to work with an extension of the endo functor to the category of classes and use the class version of the Final Coalgebra Theorem. Here we shall instead reformulate F by imposing a cardinality bound that constrains the amount of non-determinacy we allow.

We assume given a fixed infinite regular cardinal number κ and call sets of cardinality $< \kappa$ **small**.[2] We now revise the definition of F by replacing the use of the set operator *pow* by the set operator *spow*, where for each set

[2] On a first reading the reader may prefer to assume that $\kappa = \aleph_0$, so that the small sets are just the finite ones.

Y, the set $spow(Y)$ is the set of all small subsets of Y. In this way we get a bounded standard endo functor, so that we can apply the Final Coalgebra Theorem to get a final coalgebra.

So now the LTSs $(A, \xrightarrow{\mu})_{\mu \in Act}$ corresponding to the coalgebras for F have the property that $\{(\mu, a') \mid a \xrightarrow{\mu} a'\}$ is a small set for each $a \in A$.

CCS

We give a quick outline of the syntax and operational semantics of CCS, up to strong bisimulation and end with a final coalgebra characterisation. We are explicit only with the dynamic part of CCS, using dots to indicate the missing static constructors.

Syntax: The set E of CCS **agents** consist of 'expressions' e whose abstract syntax is given by the syntax equation

$$e ::= \quad \mu.e \ \mid \ \Sigma_{i \in I} e_i \ \mid \ c \ \mid \cdots$$

where $\mu \in Act$, the index set I is small and c is an **agent constant**. It is assumed that there is a set of these agent constants and with each constant c is associated a defining equation $c =_{def} e_c$, where $e_c \in E$. Any constant, including c itself, may appear in e_c, so that in general the definitions will be mutually recursive. It is possible to choose the constants with their defining equations in such a way that, roughly speaking, every possible agent is actually represented in E. When this is done we call the system of constants a **universal system of constants**[3]

Operational Semantics: The transition relations $\xrightarrow{\mu}$ on E, for $\mu \in Act$, are simultaneously inductively defined as the smallest relations satisfying the following rules.

$$\boxed{\begin{array}{lll} \mu.e \xrightarrow{\mu} e & \dfrac{e_j \xrightarrow{\mu} e}{\Sigma_{i \in I} e_i \xrightarrow{\mu} e} & (j \in I) \\[2ex] \dfrac{e_c \xrightarrow{\mu} e}{c \xrightarrow{\mu} e} & \cdots & \end{array}}$$

We can now make E into a coalgebra (E, θ). For each $e \in E$ let

$$\theta(e) = \{(\mu, e') \mid e \xrightarrow{\mu} e'\}.$$

Exercise 4.10 *Show that $\theta(e)$ is always a small set.*

[3] See [3] for a way to make this precise

Strong Bisimulation: The transition relations on E give us the computation steps for a notion of computation where the states of the computation are the agents in E. What should it mean to assert that agents e_1, e_2 have the same observable computational behaviour, given that only the atomic actions are observable? The crudest possible answer is that it should mean that they have the same trace sets; i.e. the same strings of atomic actions should label the computations from e_1 as from e_2. But this answer does not take account of any relationship between the corresponding intermediate states of the computations. It is natural to want the corresponding intermediate states to also have the same observable computational behaviour.

Milner's notion of strong bisimulation is intended to capture this idea. **Strong Bisimulation** on E can be characterised as the largest equivalence relation,$\sim$, on E such that if $e_1 \sim e_2 \xrightarrow{\mu} e_2'$ then $e_1 \xrightarrow{mu} e_1' \sim e_2'$ for some $e_1' \in E$. Let $E/\!\sim$ be the quotient of E with respect to $\sim$; i.e. it is the set of equivalence classes $[e] = \{e' \mid e \sim e;\}$ for $e \in E$. We may make this set $E/\!\sim$ into a quotient coalgebra $(E/\!\sim, \theta_\sim)$ where

$$\theta_\sim([e]) = \{(\mu, [e']) \mid e \xrightarrow{\mu} e'\}$$

for each $[e] \in E/\!\sim$.

Exercise 4.11 *Show that $\theta_\sim : E/\!\sim \;\rightarrow spow(Act \times E/\!\sim)$ is well defined and that if $\pi(e) = [e]$ for each $e \in E$ then π is a coalgebra homomorphism $(E, \theta) \rightarrow (E/\!\sim, \theta_\sim)$.*

Now let (P, η) be a final coalgebra. So $\eta : P \rightarrow spow(Act \times P)$. We are now ready to state our concluding result.

Theorem: 4.12

1. *Let $[[\;]] : (E, \theta) \rightarrow (, \eta)$ be the unique coalgebra homomorphism. Then*

$$[[e_1]] = [[e_2]] \;\Leftrightarrow\; e_1 \sim e_2.$$

2. *If a universal system of constants is used in defining E then the function $[[\;]] : E \rightarrow P$ is surjective, so that the quotient coalgebra $(E/\!\sim, \theta_\sim)$ is isomorphic to (P, η) and so is itself a final coalgebra.*

The theorem shows how to construct a final coalgebra as a quotient of the set of CCS agents by the strong bisimulation equivalence relation. In fact this construction can be taken to be the prototype for the general construction of a final coalgebra for a bounded standard endo functor needed to prove the Final Coalgebra Theorem.

Further Reading

The book [4] is a useful introduction to category theory . The first 59 pages cover the category theory needed in these lectures. The book [5] is one of several that will give a good general introduction to the topic of the semantics of programming languages. In the fourth lecture there is a very brief look at CCS. There is now a voluminous literature on CCS and other approaches to process algebra and concurrency. The original book on CCS is [6]. The final universe approach to CCS described here was first presented in [1] and developed further in [3]. Final coalgebra theorems appear in [1, 2, 3].

References

[1] Aczel, P.: *Non-Well-Founded Sets.* CSLI Lecture Notes, Number 14, Stanford University 1988.

[2] Aczel, P., Mendler, P.: **A Final Coalgebra Theorem**. in: *Proceedings of the 1989 Summer Conference on Category Theory and Computer Science*, Springer Lecture Notes in Computer Science vol.389, September, 1989, pp. 357-365.

[3] Aczel, P.: **Final Universes of Processes**. Springer Lecture Notes in Computer Science, Vol. 802, 9th MFPS Conference proceedings, edited by S. Brookes et al, 1994 pp 1-28.

[4] Barr, M., Wells, C.: *Category Theory for Computing Science.* Prentice Hall, 1990, 2nd edition 1995.

[5] Winskell, G.: *The Formal Semantics of Programming Languages.* MIT Press, 1993.

[6] Milner, R.: *Communication and Concurrency.* Prentice Hall, 1989.