Categories for The Working Estonian

Neil Ghani

Arula, Estonia

February 19, 2003

- **Aims:** I hope to achieve the following
  - Give you a feeling of why people rave about category theory
  - Familiarise you with the basic definitions and structures
  - Bolster this by examples from computer science
  - Have fun, get you to think and to teach you something.
- **However:** I don't intend to
  - Give a systematic development of the subject
  - Give any detailed proofs or discuss intricate areas

Structure of the Course

- **Lecture 1:** Motivations and Basic Definitions. Categories, Functors and Natural Transformations. The Yoneda lemma

- **Lecture 2:** Universal properties, Representability, Limits, Colimits and Adjunctions

- **Lecture 3:** Semantics of Lambda Calculi, CCC's and Fibrations

- **Lecture 4:** Datatypes, Initial Algebras and Monads. Haskell

- **Lecture 5:** Transition Systems, Bisimulation and Coalgebra

I: Categories, Functors and Natural Transformations

Neil Ghani

Arula, Estonia

February 19, 2003

Categories as Models of Set Theory

- **Motivation:** Set theory is widely regarded as a foundation of mathematics. But which set theory
  - Classical ZF set theory
  - ZF extended with various independent examples
  - Intuitionistic and Constructive set theory
- **Categories:** Provide an abstract model of set theory
  - Take as primitive the existence of some sets, functions, the ability to compose functions and a unit for composition.
- **Advantages:** Categorical models are more abstract, closer to how we do mathematics and are more algebraic

- **Motivation:** As an undergraduate, I studied
  - Group Theory, Ring Theory, Division rings, fields etc
  - We kept proving the same things, eg the existence of free structures, products of structures etc
- **Categories:** Provide a uniform meta-theory of algebra
  - Take as primitive the existence of models of an algebraic theory, homomorphisms of such structures, the ability to compose homomorphisms and a unit for composition.
- **Advantages:** Categorical models of universal algebra allow us to deal simultaneously with all examples.

- **Motivation:** CS has spawned a proliferation of logics
  - Classical, intuitionistic, constructive
  - Modal, temporal, linear, impredicative
- **Categories:** Provide a uniform meta-theory of such logics
  - Take as primitive assumptions and the proofs between them together with the ability to compose/cut proofs
- **Advantage:** Prove abstract results about classes of logics and understand the relationships between logics, eg between intuitionistic and linear logic.

- **Motivation:** There are many different programming languages
  - Imperative, functional, logic and OO-programming languages
  - Within each paradigm there are many examples
- **Categories:** Provide a uniform meta-theory of such languages
  - Take as primitive resources, programs which consume resources to produce other resources and the ability to compose such programs
- **Advantages:** We can discuss the nature of computation in a manner independent of specific languages but at a higher level than Turing machines. And we don't have to hunt the perfect domain!

- **Motivation:** Of course all 4 of these motivations interact, programs as proofs, logical definability in algebra etc.

- **Definition:** Each of these examples consist of

  - Objects: A class $\mathcal{C}$ of objects, eg *sets*, *groups*, *contexts*, *domains*

  - Morphisms: For each pair of objects $X, Y \in \mathcal{C}$, a class of morphisms $\mathcal{C}(X,Y)$, eg *functions, homomorphisms, proofs, programs*

  - Composition: For each triple $X, Y, Z \in \mathcal{C}$, a composition operation $\circ : \mathcal{C}(Y,Z) \times \mathcal{C}(X,Y) \longrightarrow \mathcal{C}(X,Z)$ which is associative.

  - Units: For each object $X \in \mathcal{C}$, a morphism $1_X \in \mathcal{C}(X,X)$ such that $f \circ 1_X = f = 1_X \circ f$

- **Fundamental Idea:** Form categories consisting of mathematics objects and the structure preserving maps between them.

- **Examples:**

  – The category **Sets** of sets and functions between them.

  – The category **Grp** of groups and group homomorphisms.

  – The category **Pos** of posets and monotone functions.

  – The category **Top** of topological spaces and cont. functions

  – The category of **CPO** of cpo's and cont. functions

  – The category **Gph** of graphs and graph homomorphisms.

- **Fundamental Idea:** Study structures by studying the structure preserving morphisms between them
  - Compare with the idea of abstract datatypes

- **Question:** So what is a morphism of categories

- **Functors:** A functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ consists of
  - A map $F : \mathcal{C} \longrightarrow \mathcal{D}$ acting on objects
  - For each pair of objects $X, Y \in \mathcal{C}$, a map $F_{X,Y} : \mathcal{C}(X, Y) \longrightarrow \mathcal{D}(FX, FY)$
  - Such that composition and identities are preserved, that is $F(f \circ g) = F(f) \circ F(g)$ and $F(1_X) = 1_{FX}$

Examples of Functors — Pay Attention Now!

- **Forgetful Functors:** The functor $U : \mathbf{Grp} \longrightarrow \mathbf{Sets}$ returns the underlying set of a group. It *forgets* the extra structure

- **Free Functors:** The functor sending a set X to the set X^* of sequences of elements of X constructs the *free* monoid on X.

- **Polymorphic Datatypes:** The functor $\mathrm{List} : \mathbf{Sets} \longrightarrow \mathbf{Sets}$ mapping a set X to the set of lists of elements of X

- **Constant Functors:** If $X \in \mathcal{D}$, then the functor $K_X : \mathcal{C} \longrightarrow \mathcal{D}$ maps each objects of $\mathcal{C}$ to X

- **Hom Functors:** If $X \in \mathcal{C}$, then $\mathcal{C}(\_, X) : \mathcal{C}^{op} \longrightarrow \mathbf{Sets}$ maps an object Y to the set of morphisms $\mathcal{C}(Y, X)$

- **Categories of categories:** As with other algebraic structures,
  - The collection of all *small* categories and functors forms a category **Cat**

- **Proof:** We have our objects and morphisms.
  - Define composition.
  - Define the identity morphism.
  - Show that
    * Composition is associative
    * The identity functor is a unit for composition

- **Example 1:** For each set X, $[]_X : X \longrightarrow \mathbf{List}(X)$ maps x to the singleton list $[x]$. For every $f : X \longrightarrow Y$

$$\begin{array}{ccc} X & \xrightarrow{[]_X} & \mathbf{List}(X) \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathbf{List}(f)} \\ Y & \xrightarrow[{[]_Y}]{} & \mathbf{List}(Y) \end{array}$$

- **Example 2:** For each set X, $hd_X : \mathbf{List}(X) \longrightarrow 1 + X$ maps empty lists to 1 and others to their head. For every $f : X \longrightarrow Y$

$$\begin{array}{ccc} \mathbf{List}(X) & \xrightarrow{hd_X} & 1 + X \\ {\scriptstyle \mathbf{List}(f)}\downarrow & & \downarrow{\scriptstyle 1+f} \\ \mathbf{List}(Y) & \xrightarrow[hd_Y]{} & 1 + Y \end{array}$$

- **Definition:** Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\alpha : F \to G$ consists of a family of maps $\alpha_X : FX \to GX$ in $\mathcal{D}$ for each $X \in \mathcal{C}$ such that for every $f : X \longrightarrow Y$

$$\begin{tikzcd} FX \arrow[r, "\alpha_X"] \arrow[d, "Ff"'] & GX \arrow[d, "Gf"] \\ FY \arrow[r, "\alpha_Y"'] & GY \end{tikzcd}$$

- **Example 1:** Take $\mathcal{C} = \mathcal{D} = \textbf{Sets}$. F is the identity functor and G is the $\mathbf{List}$ functor

- **Example 2:** Take $\mathcal{C} = \mathcal{D} = \textbf{Sets}$. F is the $\mathbf{List}$ functor and G is the functor mapping a set X to the set $1 + X$

Functor Categories

- **Motivation:** A rich source of constructing interesting categories

- **Definition:** Fix categories $\mathcal{C}$ and $\mathcal{D}$. The functor category $[\mathcal{C}, \mathcal{D}]$ has

  - Objects: These are functors $F : \mathcal{C} \longrightarrow \mathcal{D}$

  - Morphisms: Given functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$, the homset $[\mathcal{C}, \mathcal{D}](F, G)$ consists of all natural transformations form F to G

- **Proof:** We must

  - Define the composition of natural transformations, define an identity and prove associativity etc.

- **Fundamental Idea:** We reason about objects by reasoning about the morphisms between them.
  - Encapsulated by sating for each object $X \in \mathcal{C}$, we have the hom functor $\mathcal{C}(\_, X) : \mathcal{C}^{op} \longrightarrow \mathbf{Sets}$
  - Each $f : X \longrightarrow Y$, defines a natural transformation of the form $f \circ \_ : \mathcal{C}(\_, X) \longrightarrow \mathcal{C}(\_, Y)$
- **Lemma:** This defines a functor $Y : \mathcal{C} \longrightarrow \mathbf{Sets}^{\mathcal{C}^{op}}$
- **Lemma:** The functor Y is full and faithful
  - The intuition that morphisms are the key, formalised by hom-functors, is correct. This justifies the definition of natural transformations.

II: Representability, Universal Properties, Limits and Colimits, Adjunctions

Neil Ghani

Arula, Estonia

February 19, 2003

- **Motivation:** Categories are an algebraic structure which consists of objects and morphisms between them
  - Applications include set theory, algebra, logic and program language semantics

- **Fundamental Ideas:** We have seen two fundamental ideas
  - Mathematical objects and their structure preserving maps tend to form categories.
  - We study the properties of objects by studying their morphisms.

- **The Category Cat:** Maps between categories are called functors. Maps between functors are called natural transformations

- **Motivation:** In Lecture 1 we looked at the world outside a category.
  - Lecture 2 looks at structure inside a category

- **Objectives:** To cover
  - *Representability:* Internalising external structure inside a category
  - *Universal properties:* A specification mechanism for objects
  - *Limits and Colimits:* An example of universal properties
  - *Adjunctions:* Possibly the most important tool of a category theorist.

A Running Example — Products

- **Question:** What is the cartesian product of two sets
  - $A \times B = \{ (x, y) \mid x \in A,\ y \in B\}$

- **But:** We can argue that the above is not a good answer
  - It does not tell us what the essence of a product is. This is important if we want to take about the product of groups, rings, datatypes, logical propositions etc in a uniform manner
  - It makes use of an internal representation of the product - cf abstract datatypes in CS. No reasoning principle.
  - The product should be abstract concept with a reasoning principle which applies to any category. We don't want a specific representation as this would be tied to a specific category.

- **Definition:** Let $\mathcal{C}$ be a category. The product of two objects $X, Y \in C$ is:
  - An object $X \times Y$ and maps $\pi_1 : X \times Y \longrightarrow X$ and $\pi_2 : X \times Y \longrightarrow Y$, such that
  - For any other object Z and maps $f : Z \longrightarrow X$, $g : Z \longrightarrow Y$, there is a unique map $h : Z \longrightarrow X \times Y$ such that:

$$\begin{tikzcd} & Z \arrow[dl, "f"'] \arrow[d, "h"] \arrow[dr, "g"] & \\ X & X \times Y \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & Y \end{tikzcd}$$

- **Advantages:** Applicable to any category by giving a specification in terms of morphisms. Captures the essence of the product which is what we want to reason with. Hides implementation.

So, What is Going on Here?

- **Motivation:** We want to understand this form of construction and reasoning and use it elsewhere. Fix X and Y
  - Call an object Z and maps $f : Z \longrightarrow X$ and $g : Z \longrightarrow Y$ a cone. This defines a functor $cone : \mathcal{C}^{op} \longrightarrow$ **Sets**
  - The product represents this external construction inside the category as morphisms into $X \times Y$

- **Representability:** A functor $F : \mathcal{C}^{op} \longrightarrow$ **Sets** is *representable* iff there is an object X such that $F \cong \mathcal{C}(\_, X)$
  - The functor $cone$ is represented by the object $X \times Y$.

- **Fundamental Idea:** We are representing external structure inside the category using the morphisms of the category.

- **Definition:** Let $F : \mathcal{C} \longrightarrow \mathcal{D}$ be a functor and $d \in \mathcal{D}$. A universal arrow from F to d is
  - An object $c \in \mathcal{C}$ and an arrow $f : Fc \rightarrow d$ such that
  - For all other arrows $g : Fc' \rightarrow d$, there is a unique $h : c \rightarrow c'$ st.

$$
c' \xrightarrow{\;} \quad
\begin{tikzcd}
c' \arrow[d, "h"'] \\
c
\end{tikzcd}
\qquad
\begin{tikzcd}
Fc' \arrow[d, "Fh"'] \arrow[dr, "g"] & \\
Fc \arrow[r, "f"'] & d
\end{tikzcd}
$$

- **Intuition:** So c is somehow the closest object in $\mathcal{C}$ to d via F

- **Lemma:** c is a universal arrow from F to d, iff the functor $\mathcal{D}(F\_, d) : \mathcal{C}^{op} \longrightarrow$ **Sets** is representable

- **Uses:** Universal properties are a way of specifying an object by saying i) it has some arrows; and ii) any other object with similar arrows maps into it
  - For the product let $\Delta : \mathcal{C} \longrightarrow \mathcal{C}^2$ be the diagonal functor. The projections π_1, π_2 are universal from Δ to X, Y

- **Definability:** Does specification by universality actually produce a valid definition.
  - *No:* Many objects which satisfy universal properties. For example $\{(y, x) \mid x \in X,\ y \in Y\}$ is a product $X \times Y$
  - *Yes:* But they are all isomorphic. So, what is important is not how we represent them, but the universal property that defines them. Again, this is abstract datatypes!

- **Example:** An equaliser for a pair of arrows $f, g : X \to Y$ is
  - An arrow $h : Z \to X$ such that $fh = gh$ such that
  - For any other arrow $h' : Z' \to X$ such that $fh' = gh'$, there is a unique arrow $k : Z' \to Z$ such that $h' = hk$

$$\begin{tikzcd} Z \arrow[r, "h"] & X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & Y \\ Z' \arrow[u, "k"] \arrow[ur, "h'"'] & & \end{tikzcd}$$

- **Concretely:** The equaliser in **Sets** is the kernel of f and g

- **Universality:** We postulate the existence of an arrow h which has a property and require any other arrow h' satisfying the same property to factorise through h

- **Motivation:** A generalisation of products and equalisers
  - A diagram in a category $\mathcal{C}$ is a functor $F : \mathcal{D} \to \mathcal{C}$. We usually just draw the image of F inside $\mathcal{C}$.
  - A cone for a diagram $F : \mathcal{D} \to \mathcal{C}$ is an object $c \in \mathcal{C}$ and natural transformation $\gamma : K_c \longrightarrow F$. That is, for each object $d \in D$ an arrow $\gamma_d : c \to Fd$ such that the triangles commute.
- **Definition:** The limit of $F : \mathcal{D} \to \mathcal{C}$ is the universal cone. That is, $limF$ is a cone through which all other cones factorise
- **Example:** There are two more canonical limits
  - *Terminal objects:* Take $\mathcal{D}$ to be the empty category
  - *Pullbacks:* Take $\mathcal{D}$ to be the category $\bullet \longrightarrow \bullet \longleftarrow \bullet$

- **Definition:** If $\mathcal{C}$ is a category, then $\mathcal{C}^{op}$ is the category with
  - *Objects:* $\mathcal{C}^{op}$ has the same objects as $\mathcal{C}$
  - *Morphisms:* $\mathcal{C}^{op}(X,Y) = \mathcal{C}(Y,X)$
  - *Composition etc:* All inherited from $\mathcal{C}$

- **Uses:** Although duality is simple, it is powerful
  - We have already used $\mathcal{C}^{op}$ to deal with contravariant functors
  - More generally, whenever we have a construction in a category $\mathcal{C}$, interpret the construction in $\mathcal{C}^{op}$ and get the dual construction. Proofs about the construction also dualise

- **Application:** The dual of a limit is a *colimit*
  - *Coproducts, coequalisers, pushouts, initial objects*.

- **Example:** The coproduct of two objects $X, Y \in \mathcal{C}$ is
  - An object $X + Y$ and arrows $in_1 : X \to X + Y$, $in_2 : Y \to X + Y$
  - For any other object Z and arrows $f : X \to Z$, $g : Y \to Z$, there is a unique arrow h making the triangles commute.

$$\begin{tikzcd} & Z & \\ X \arrow[ur, "f"] \arrow[r, "in_1"'] & X + Y \arrow[u, "h"] & Y \arrow[ul, "g"'] \arrow[l, "in_2"] \end{tikzcd}$$

- **Duality:** The factorising arrow goes out of the colimit

- **Fundamental Idea:** Adjoints are possibly the most important tool at a category theorist's disposal

- **Example 1:** Recall the diagonal functor $\Delta : \mathcal{C} \longrightarrow \mathcal{C}^2$. The universal property of products can be stated as a bijection of arrows

$$\mathcal{C}^2(\Delta Z, (X, Y)) \cong \mathcal{C}(Z, X \times Y)$$

- **Example 2:** Let U : **Mon** $\longrightarrow$ **Sets** be the forgetful functor sending a monoid to its carrier set. Then the free monoid X^* on a set X satisfies the bijection of arrows

$$\mathbf{Mon}(X^*, M) \cong \mathbf{Sets}(X, UM)$$

- **Motivation:** We can generalise these examples

- **Definition:** Let $F : \mathcal{C} \longrightarrow \mathcal{D}$ and $G : \mathcal{D} \longrightarrow \mathcal{C}$. F is left-adjoint to G, written $F \dashv G : \mathcal{C} \to \mathcal{D}$ iff there is a natural bijection

$$\mathcal{D}(FX, Y) \cong \mathcal{C}(X, GY)$$

- **Examples:** In our two examples

  – The product functor $\_ \times \_ : \mathcal{C}^2 \longrightarrow \mathcal{C}$ is right adjoint to the diagonal functor $\Delta : \mathcal{C} \longrightarrow \mathcal{C}^2$.

  – The free monoid functor F_M is left adjoint to the forgetful functor U : **Mon** $\longrightarrow$ **Sets**

- **Limits and Colimits:** Limits and colimits are adjoints.
  - Given a category $\mathcal{D}$, the diagonal functor can be generalised to $\Delta : \mathcal{C} \longrightarrow \mathcal{C}^{\mathcal{D}}$ where $\Delta(X) = K_X$ is the constant functor. Then $colim \dashv \Delta \dashv lim$

$$\mathcal{C}(colimF, X) \cong \mathcal{C}^{\mathcal{D}}(F, K_X) \quad \text{and} \quad \mathcal{C}^{\mathcal{D}}(K_X, F) \cong \mathcal{C}(X, limF)$$

- **Free Structures:** Given any algebraic theory A, eg groups, monoids etc there is a forgetful functor $U_A : Mod_A \to \mathbf{Sets}$
  - Free algebras are left adjoint $F_A \dashv U_A : \mathbf{Sets} \longrightarrow Mod_A$

$$Mod_A(F_A X, M) \cong \mathbf{Sets}(X, UM)$$

- **Motivation:** Link adjoints to representability, universal arrows etc

- **Lemma:** There is an adjunction $F \dashv G : \mathcal{C} \rightarrow \mathcal{D}$ or $\mathcal{C}(c, Gd) \cong \mathcal{D}(Fc, d)$ iff
  - For every d, the functor $\mathcal{D}(F\_, d) : \mathcal{C}^{op} \rightarrow$ **Sets** is representable
  - For every d, there is a universal arrow from F to d

- **(Co-)Unit:** Thus there are natural arrows $c \rightarrow GFc$ and $FGd \rightarrow d$

- **Theorem:** Adjoint functors satisfy many properties
  - Left adjoints preserve colimits, right adjoints preserve limits

III: Semantics of Lambda Calculi, CCC's and Fibrations

Neil Ghani

Arula, Estonia

February 19, 2003

Recall

- **Categories:** Categories provide a uniform model of set theory, algebra, logic and computation
  - As primitive we take some object and for each pair of objects a collection of morphisms which can be composed
  - Allied concepts are functors and natural transformations
- **Internal Structure:** Representability is a key concept which uses morphisms to internalise structure inside a category.
  - Universal arrows satisfy certain properties. Any other arrow satisfying the property factorises through the universal arrow
  - Examples are limits and colimits
  - All of this leads to the fundamental concept of an adjunction $F \dashv G : \mathcal{C} \longrightarrow \mathcal{D}$ where $\mathcal{C}(Fd, c) \cong \mathcal{D}(d, Gc)$

Overview of Lecture 3:

- **Aims:** Apply category theory to progam language semantics
  - *Logic:* This is also a semantics for intuitionistic logic via the *propositions as types* paradigm
- **Why:** Reasoning is important. Denotational semantics is more modular that operational semantics
  - When are two programs (proofs) equivalent
- **Objectives:** We study two structures
  - Church's λ-calculus which is the foundation of functional programming. We model these using *cartesian closed categories*
  - Curry's λ-calculus which adds polymorphic type variables. We model these using *fibrations*

The Simply Typed Lambda Calculus $\lambda^{1,\times,\rightarrow}$

- **Motivation:** $\lambda^{1,\times,\rightarrow}$ is a fragment of the untyped λ-calculus. Often used as a meta-language for programming languages
- **Types:** We ignore base types, eg `Nat`, `Bool` and recursion. Let V be a set of variables. The types of $\lambda^{1,\times,\rightarrow}$ are

$$\sigma := \ 1 \mid \sigma \times \sigma \mid \sigma {\rightarrow} \sigma$$

- **Contexts:** Contexts are of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ where $x \in V$ and σ_i are types. Variables aren't repeated etc
- **Pre-terms:** Defined inductively by

$$t := \ x \mid * \mid \langle t, t \rangle \mid \pi_1 t \mid \pi_2 t \mid \lambda x{:}\sigma.\, t \mid t\ t$$

- **Judgements:** If Γ is a context, t a pre-term and σ a type, judgements are of the form $\Gamma \vdash t : \sigma$

Inductive definition of $\lambda^{1,\times,\to}$ Judgements

- **Structural Rules:** Allow variables to be used arbitrarily

$$\frac{(x:\sigma)\in\Gamma}{\Gamma\vdash x:\sigma} \qquad \frac{\Gamma,x:\sigma,y:\sigma\vdash t:\sigma'}{\Gamma,z:\sigma\vdash t[z/x,z/y]:\sigma'} \qquad \frac{\Gamma\vdash t:\sigma'}{\Gamma,x:\sigma\vdash t:\sigma'}$$

- **Logical Rules:** For forming and using terms of compound type

$$\frac{\Gamma,x:\sigma\vdash t:\sigma'}{\Gamma\vdash \lambda x{:}\sigma.\,t:\sigma\to\sigma'} \qquad \frac{\Gamma\vdash t:\sigma_1\times\sigma_2}{\Gamma\vdash \pi_i t:\sigma_i} \qquad \frac{}{\Gamma\vdash *:1}$$

$$\frac{\Gamma\vdash t:\sigma\to\sigma' \quad \Gamma\vdash u:\sigma}{\Gamma\vdash t\;u:\sigma'} \qquad \frac{\Gamma\vdash t:\sigma_1 \quad \Gamma\vdash u:\sigma_2}{\Gamma\vdash\langle t,u\rangle:\sigma_1\times\sigma_2}$$

- **Observations:** $\lambda^{1,\times,\to}$ is a fragment of untyped λ-calculus
  - $\lambda^{1,\times,\to}$ defines proofs for intuitionistic propositional logic
  - Change the structural and product rules to get linear logic

- **Motivation:** When do two terms define the same function or proof? Answer this by modelling $\lambda^{1,\times,\to}$ in a category $\mathcal{C}$.
  - That is, define $[\![.]\!] : \ \lambda^{1,\times,\to} \longrightarrow \mathcal{C}$

- **Types:** Each type will be modelled by an object of $\mathcal{C}$. $[\![1]\!] = 1$ and $[\![\sigma \times \sigma']\!] = [\![\sigma]\!] \times [\![\sigma']\!]$

- **Contexts:** Each context modelled as a product of its types

$$[\![x_1 \!:\! \sigma_1, \ldots, x_n \!:\! \sigma_n]\!] = [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$$

- **Terms:** Each judgement will be interpreted as

$$[\![\Gamma \vdash t : \sigma]\!] \text{ is a morphism } [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash t:\sigma]\!]} [\![\sigma]\!]$$

- **Structure:** $\mathcal{C}$ must have finite products

- **So far:** Category theory models contexts. Now use induction to model judgements

- **Structural Rules:** Projection models structural rules

$$
\begin{aligned}
[\![\Gamma \vdash x : \sigma]\!] &= [\![\Gamma]\!] \xrightarrow{\pi} [\![\sigma]\!] \\
[\![\Gamma, x : \sigma \vdash t : \sigma']\!] &= [\![\Gamma]\!] \times [\![\sigma]\!] \xrightarrow{\pi} [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash t : \sigma']\!]} [\![\sigma']\!]
\end{aligned}
$$

- **Unit Type:** $*$ is modelled by the terminal object

$$[\![\Gamma \vdash * : 1]\!] = [\![\Gamma]\!] \xrightarrow{!} 1$$

- **Product Types:** Modelled by the product in the category

  – Projections are modelled by projections, $\langle t, u \rangle$ by the factoring arrow.

- **Motivation:** How do we account for function spaces
  - Assume an operation $\to$ on objects. Then $[\![\sigma \to \sigma']\!] = [\![\sigma]\!] \to [\![\sigma']\!]$
- **Question:** What structure do we want on $\to$?

$$\mathcal{C}([\![\Gamma]\!] \times [\![\sigma]\!], [\![\sigma']\!]) \underset{[\![\_\ x]\!]}{\overset{[\![\lambda x.\_]\!]}{\rightleftarrows}} \mathcal{C}([\![\Gamma]\!]\ ,\ [\![\sigma]\!] \to [\![\sigma']\!])$$

- **More Abstractly:** An isomorphism $\mathcal{C}(X \times Y, Z) \cong \mathcal{C}(X, Y \to Z)$
- **Even More Abstractly:** $\_ \times Y \dashv Y \to \_ : \mathcal{C} \longrightarrow \mathcal{C}$
- **Observation:** This is just applying categorical tools
  - And it works!

- **Definition:** A category $\mathcal{C}$ is cartesian closed iff
  - $\mathcal{C}$ has finite products (a property of adjoints)
  - For each object X, the functor $\_ \times X$ has a right adjoint
- **Recall:** Adjoints are given as universal arrows
  - That is, some arrows exist and all other such arrows factorise through the universal arrow
- **Introduction and Elimination:** These rules are not arbitrary
  - Elimination gives the universal arrow. Introduction is the factorising arrow
  - That introduction is a cone is β-equality. Uniqueness is η-equality

Deficiencies in $\lambda^{1,\times,\to}$

- **Syntactic Deficiencies:** $\lambda^{1,\times,\to}$ is a basic calculus and as such suffers from
  - We like polymorphism
  - We like datatypes
  - We like dependent datatypes
  - We like recursion

- **Semantic Deficiencies:** Type theoretically, contexts are not types
  - Categorically, we used products to model both contexts and the product type

- **Motivation:** How can we understand the type `Vec(n)` of vectors of length `n`
  - Note this type contains a term within it so we cant just define types and then terms
  - Dependent type theory is rather complicated!

- **Indexed Categories:** Separate contexts from types
  - Define a category $\mathcal{B}$ of contexts and context morphisms
  - And a functor sending each context I to the category $\mathcal{E}_I$ of types and terms definable in that context. $\mathcal{E}_I$ is called the *fibre* over I
  - Functoriality (Tarmo keep quiet!) is substitution

- **Motivation:** Indexed categories $F : \mathcal{B}^{op} \longrightarrow \mathbf{Cat}$ capture the idea of a category depending upon/varying over $\mathcal{B}$

- **Fibrations:** Group all the fibres together in one category $\mathcal{E}$
  - Take as primitive the projection functor $p : \mathcal{E} \longrightarrow \mathcal{B}$

- **Detail:** Define the categories $\mathcal{B}$ and $\mathcal{E}$ as follows
  - $\mathcal{B}$ to have as objects contexts Γ and arrows $\Gamma \to \Delta$ judgements $\Gamma \vdash t_i : \Delta_i$.
  - $\mathcal{E}$ has as objects type judgements $\Gamma \vdash \sigma\ type$ and arrows $(\Gamma \vdash \sigma\ type) \longrightarrow (\Delta \vdash \tau\ type)$ a context morphism $\Gamma \vdash t_i : \Delta_i$ and a judgement $\Gamma, x : \sigma \vdash t : \tau[t_i/y_i]$

- **Fibres:** The indexed category approach can be recovered
  - $\mathcal{E}_I$ has objects $E \in \mathcal{E}$ st $pE = I$ and arrows $f : E_1 \to E_2$ st $pf = 1_I$

- **Crucial Idea:** Every arrow in $\mathcal{E}$ contains two components
  - A *cartesian* part which is the lifting of a $\mathcal{B}$ arrow. p projects this *cartesian* part out
  - A *vertical* part which is its $\mathcal{E}$-component
  - $\mathcal{E}$ arrows are the composite of a vertical and cartesian arrow
  - But what is a cartesian arrow.

- **Why Fibrations?** We give a universal property for cartesian arrows. This is a reasoning principle which is left implicit in indexed categories

- **Definition:** An arrow $f : E \to D$ is cartesian iff for every arrow $f' : E' \to D$ such that $pf' = pf \circ v$, there is unique map $! : E' \to E$ over v such that $f' = f \circ !$.

- **Definition:** A functor $p : \mathcal{E} \longrightarrow \mathcal{B}$ is a fibration iff for every arrow $u : I \to pE$ there is a cartesian arrow over u.

- **Lemma:** In a fibration, every $\mathcal{E}$-arrow factorises as a vertical arrow composed with a cartesian arrow.

  – Let $f : E \to D$. Consider $pf : pE \to pD$. Let k be cartesian over pf. Clearly $pf = pf \circ 1$. Thus there is a v such that $f = k \circ v$

Generic Objects

- **So far:** I separated contexts from product types and modelled dependent types in a fibration.
  - What about polymorphism?

- **Types:** Simply add type variables
  - No quantification over type variables as in System F

- **Definition:** Let $p : \mathcal{E} \longrightarrow \mathcal{B}$ be a fibration. An object $T \in \mathcal{E}$ is a *generic object* iff for every object E, there is a cartesian morphism $E \rightarrow T$

IV: Semantics of Datatypes, Initial Algebras and Final Coalgebras

Neil Ghani

Arula, Estonia

February 19, 2003

- **Categories:** Categories provide a uniform model of set theory, algebra, logic and computation
  - As primitive we take some object and for each pair of objects a collection of morphisms which can be composed
  - Allied concepts are functors and natural transformations
- **Representability:** Internalises structure within a category
  - Universal arrows satisfy certain properties. Any other arrow satisfying the property factorises through the universal arrow
  - The fundamental concept of an adjunction $F \dashv G : \mathcal{C} \longrightarrow \mathcal{D}$ where $\mathcal{C}(Fd, c) \cong \mathcal{D}(d, Gc)$
- **Applications:** Semantics of type theory, CCC's etc

- **Aims:** Extend type theory to cover datatypes
  - Recursive datatypes, eg `Nat` of natural numbers
  - Polymorphic datatypes, eg `Tree a` stores data of any type `a`
  - Higher order datatypes, eg `Lam a` of α-equivalence classes of untyped λ-terms
- **Objectives**
  - Explain introduction and eliminations rules,
  - Define semantic equalities
  - Explain pattern matching

- **Motivation:** Assume we have some semantic category $\mathcal{C}$ modelling basic types. What structure does $\mathcal{C}$ require to model datatypes.

- **Datatypes:** Datatypes are defined by giving constructors and their inputs

```
data Nat      =   Zero | Succ Nat

data Tree a   =   Leaf a | Node (Tree a) (Tree a)

data Maybe a  =   Bound | Free a

data Lam a    =   Var a
              |   App (Lam a) (Lam a)
              |   Abs (Lam (Maybe a))
```

- **Motivation:** Haskell supports a convenient mechanism for defining functions. No clumsy `if` or `switch`

```
copy Zero            = []
copy (Succ x)        = 1 : copy n

flatten (Leaf x)     = [x]
flatten (Node t1 t2) = flatten t1 ++ flatten t2
```

- **Type Theory:** Datatypes given by introduction and elimination

$$\frac{}{0 : \mathsf{Nat}} \qquad \frac{n : \mathsf{Nat}}{s\ n : \mathsf{Nat}} \qquad \frac{\Gamma \vdash u : X \qquad \Gamma \vdash v : X \to X}{\Gamma \vdash It(u,v) : \mathsf{Nat} \to X}$$

- **In general:** Pattern matching and generalises to other Haskell datatypes using `fold`. I want to understand all of these concepts semantically

- **Key Idea:** The introduction and elimination rule defining `Nat` can be captured by a universal property

$$
\begin{tikzcd}
1 \arrow[r, "0"] \arrow[dr, "u"'] & Nat \arrow[d, "It(u,v)"] & Nat \arrow[l, "s"'] \arrow[d, "It(u,v)"] \\
& X & X \arrow[l, "v"]
\end{tikzcd}
$$

- **Universality:** As usual, introduction rules define some arrows
  - Elimination says other such arrows induce a factorisation
  - β-equality is the commuting property of of the mediating morphism. η-equality is the uniqueness of the mediating morphism

- **Algebras:** We can rewrite the universal property

$$
\begin{tikzcd}
1 + Nat \arrow[r, "{[0,s]}"] \arrow[d, "{It(u,v)}"'] & Nat \arrow[d, "{It(u,v)}"] \\
1 + X \arrow[r, "{[u,v]}"'] & X
\end{tikzcd}
$$

- **Definition:** If $F : \mathcal{C} \longrightarrow \mathcal{C}$ is a functor, the category F-Alg is
  - Objects are arrows $FX \xrightarrow{f} X$
  - Morphisms from $FX \xrightarrow{f} X$ to $FY \xrightarrow{g} Y$ are arrows $h : X \rightarrow Y$ such that

$$
\begin{tikzcd}
FX \arrow[r, "f"] \arrow[d, "Fh"'] & X \arrow[d, "h"] \\
FY \arrow[r, "g"'] & Y
\end{tikzcd}
$$

- **Definition:** An initial object is the colimit of the empty diagram. Or, 0 is initial iff for every object X, there is one arrow $0 \to X$. In **Sets** this is the empty set.

- **Lemma:** A natural numbers object is the initial algebra of the functor $FX = 1 + X$

- **Lemma:** An initial algebra $FX \xrightarrow{f} X$ is an isomorphism

$$\begin{tikzcd} FX \arrow[r, "Fh"] \arrow[d, "f"'] & FFX \arrow[r, "Ff"] \arrow[d, "Ff"] & FX \arrow[d, "f"] \\ X \arrow[r, "h"'] & FX \arrow[r, "f"'] & X \end{tikzcd}$$

$f \circ h = 1$ by uniqueness. $h \circ f = Ff \circ Fh = F(f \circ h) = F1 = 1$

So When do Initial Algebras Exist

- **Key Idea:** Given a functor $F : \mathcal{C} \longrightarrow \mathcal{C}$, we want the least solution of $FX \cong X$.

- **Recall:** Assume a *partial order* P has a least element $\perp$ and least upper bounds of ω-chains. If a monotone function $f : P \to P$ preserves these lubs, then the least fixed point is $\mu f = \bigvee f^i \perp$

- **Generalise:** Assume a category $\mathcal{C}$ an initial object 0 and colimits of ω-chains. If a functor $F : \mathcal{C} \longrightarrow \mathcal{C}$ preserves these colimits, then the least fixed point is $\mu F = colim\ \ F^i 0$

- **Colimit Preservation:** Does $FX = 1 + X$ preserve colimits
  - Yes. Let F be polynomial, ie built from $+$ and $\times$. In a CCC, $+$ and $\times$ are left adjoint and hence preserve all colimits!

- **Recall:** Trees are defined

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

- **Initial Algebras:** For each a, define $F_a(X) = a + X \times X$. Calculate the semantics of the datatype `Tree a` as the initial algebra μF_a

$$colim \;\; 0 \xrightarrow{!} F_a 0 \xrightarrow{F_a !} F_a F_a 0 \longrightarrow \ldots$$

- **Functoriality:** We want various properties of `Tree a`
  - $\mu : [\mathcal{C}, \mathcal{C}] \longrightarrow \mathcal{C}$ is functorial. Yes - colimit chasing
  - `Tree a` is functorial in `a`. Yes, from above

- **Recall:** A datatype for α-equivalence classed of untyped lambda terms.

```
data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

- **Problem:** We cant write a functor defining the right hand side
  - The parameter, here free variables, of the polymorphism has changed.

- **Alternatively:** As an initial algebra, we want to solve the equation

$$\Lambda X \cong X + \Lambda X \times \Lambda X + \Lambda(X + 1)$$

- **Key Idea:** Lets rewrite this equation

$$
\begin{array}{rcll}
\Lambda X & \cong & X + \Lambda X \times \Lambda X + \Lambda(X+1) & \text{in } \mathcal{C} \\
 & \cong & 1X + (\delta\Lambda)X + (\Delta\Lambda)X & \text{in } \mathcal{C} \\
\Lambda & \cong & 1 + (\delta\Lambda) + (\Delta\Lambda) & \text{in } \mathcal{C}^{\mathcal{C}}
\end{array}
$$

  where

  - $\delta : \mathcal{C}^{\mathcal{C}} \longrightarrow \mathcal{C}^{\mathcal{C}}$ is $(\delta F)X = FX \times FX$
  - $\Delta : \mathcal{C}^{\mathcal{C}} \longrightarrow \mathcal{C}^{\mathcal{C}}$ is $(\Delta F)X = F(X+1)$

- **Hence:** We now have an isomorphism we can solve.

  - $\Lambda = \mu\Phi$ where $\Phi F = 1 + (\delta F) + (\Delta F)$

- **Recall:** *Duality* is the idea of turning the arrows around
  - Lets apply this to *algebras* and get *coalgebras*

- **Definition:** If $F : \mathcal{C} \to \mathcal{C}$ is a functor, F-coalg is the category
  - Objects are arrows $X \xrightarrow{f} FX$
  - Morphisms from $X \xrightarrow{f} FX$ to $Y \xrightarrow{g} FY$ are arrows $h : X \to Y$ such that

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "h"'] & FX \arrow[d, "Fh"] \\ Y \arrow[r, "g"'] & FY \end{tikzcd}$$

- **Recall:** Given a functor, thought of as a specification of a datatype, the semantics was its initial algebra
  - Does it make sense to consider the final coalgebra of the functor

- **Lemma:** Final coalgebras are greatest fixed points.
  - If $f : P \to P$ preserves greatest lower bounds, its greatest fixed point is $\nu f = \bigwedge f^i \top$
  - If $F : \mathcal{C} \longrightarrow \mathcal{C}$ is a limit preserving functor, its final coalgebra is the limit of the ω-cochain

$$\nu F = lim \quad 1 \longleftarrow F1 \longleftarrow FF1 \longleftarrow \ldots$$

- **Example 1:** Consider $FX = 1 + X$. All polynomial functors preserve connected limits.

$$1 \xleftarrow{\;!\;} 1+1 \xleftarrow{\;1+!\;} 1+(1+1) \longleftarrow \ldots$$

- **Limits:** The limit of this diagram will consist of a family of elements x_i from $(1+X)^i0$ such that the triangles commute.
  - Thus we have the natural numbers, and an extra element

- **Co-Naturals:** N^∞ is the natural numbers with an infinity
  - In general, given a functor F, if we take the final coalgebra semantics we get finite and infinite terms and instead of `folds` we get `unfolds`. See Varmo's talk

Lecture 5: Monads

Neil Ghani

Arula, Estonia

February 19, 2003

- **Monads:** What are they?
  - Monoids in the category of endofunctors!
- **Examples:** Several examples
  - Term algebras
  - Computational Monads
  - Free Monads
  - Iterative Monads
- **The Future:** Coproducts of Monads!

Key Concepts (and some notation)

- **Signature:** A signature is $\Sigma : Nat \rightarrow \mathbf{Sets}$

- **Term Algebra:** The term algebra $T_\Sigma(X)$ over variables X is

$$\frac{x \in X}{'x \in T_\Sigma(X)} \qquad\qquad \frac{f \in \Sigma(n) \quad t_1, \ldots t_n \in T_\Sigma(X)}{f(t_1, \ldots, t_n) \in T_\Sigma(X)}$$

- **Equations:** Have the form $(X \vdash l = r)$ where $l, r \subseteq T_\Sigma(X)$

- **Free Algebra:** The free algebra for $T_A(X)$ over variables X is

$$T_A(X) = T_\Sigma(X) / \sim_E$$

- **Models:** A set & interpretations of operations validating eqns.

- **Key Idea:** Aspects of term algebra
  - Free algebra construction is a functor $T_\Sigma : \mathbf{Sets} \to \mathbf{Sets}$
  - Variables are terms, ie $\eta_X : X \to T_\Sigma(X)$
  - Substitution takes terms built over terms and flattens them:

$$\mu_X : T_\Sigma(T_\Sigma(X)) \to T_\Sigma(X)$$

- **Finitariness:** All operations have finite arity

$$T_\Sigma(X) = \bigcup_{X_0 \subset X \text{ is finite}} T_\Sigma(X_0)$$

- **Abstraction:** Lets concentrate on these ideas

- **Answer:** It is a monoid in the category $[\mathcal{C}, \mathcal{C}]$ of endofunctors!

- **Internal structure:** This kind of argument is prevalent in categorical models of constructive set theory.
  - Synthetic domain theory is one such spin off

- **Another Example:** What is a ring
  - A monoid in the category of abelian groups

- **Definition:** A monad over a category $\mathcal{C}$ is a triple $\langle T, \eta, \mu \rangle$
  - An action $T\!:\!\mathcal{C}\!\to\!\mathcal{C}$, unit $\eta\!:\!1\!\Rightarrow\!T$,
  - Such that $\mu \circ T\eta = \mu \circ \eta T = 1$ and $\mu \circ T\mu = \mu \circ \mu T$.

- **Theorem:** U : **Monad**$(\mathcal{C}) \longrightarrow \mathcal{C}^{\mathcal{C}}$ has a left adjoint mapping F to the monad F^{μ} whose action is $F^{\mu}(X) = \mu Y.X + FY$

- **Compositionality:** Since left adjoints preserve colimits,

$$\mathsf{T}_{\Sigma+\Sigma'} = \mathsf{T}_{\Sigma} + \mathsf{T}_{\Sigma'}$$

- **Summary:** Monads offer a syntax free presentation of algebraic theories based on *variables*, *substitution* and *term algbera*.

Monads in Haskell

- **Motivation:** Haskell is pure functional language
  - No side effects, exception handling, non-determinism etc
  - Moggi, Wadler etc proposed monads as a modular solution
- **Key Idea:** If T is a monad, $\Gamma \vdash t : T\sigma$ is a program which may perform some kind of computation T and then return a σ
  - *Exceptions* - $TX = X + E$
  - *State* - $TX = S \rightarrow S \times X$
  - *Nondeterminism* - $TX = List(X)$
- **Crucially:** Typing distinguishes between values and computations. If a program changes the state, this is recorded.

- **Operations:** There are two fundamental operations
  - All values are trivial computations
  - There is a type-safe mechanism for composing computations

- **Definition:** Some more Haskell

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a  -> m a
```

- **Lemma:** Kleisli triples are precisely monads. Note there are equations which cannot be expressed in Haskell

How do We Combine Monads

- **Question:** Imagine we have two programs, one which raises an exception and one which changes the state. Now we combine them.
  - Which monad does the combined program live in

- **Answer:** We need a way of combining monads
  - *Monad Transformers:* Proposed by Mark Jones. The exception monad transformer and state monad transformers are

```
data ExnT m a     = ExnT m (Exn a)

data StateT s m a = StateT (s -> m (s, a))
```

- **Definition:** A monad transformer is a pointed endofunctor of **Monad**$(\mathcal{C})$.

- **Problems:** This definition does not tell you how to compute a monad transformer
  - There may be several
  - Current ones are ad-hoc
  - The combination is not symmetric

- **Use Coproducts:** Categorically, the most natural thing is to combine T and S as $T + S$
  - This is symmetric, canonical
  - And it is even a monad transformer

- **Intuition:** Consider signatures Σ_1, Σ_2

$$
\begin{aligned}
(\mathsf{T}_{\Sigma_1} + \mathsf{T}_{\Sigma_2})X = \mathsf{T}_{\Sigma_1+\Sigma_2}X = \ & X + \\
& \mathsf{T}_1 X + \mathsf{T}_2 X + \\
& \mathsf{T}_1\mathsf{T}_2(X) + \mathsf{T}_2\mathsf{T}_1(X) + \\
& \mathsf{T}_1\mathsf{T}_2\mathsf{T}_1(X) + \mathsf{T}_2\mathsf{T}_1\mathsf{T}_2(X) + \ldots
\end{aligned}
$$

- **Problem:** This doesn't work as variables etc repeated

- **Solution:** A variety of solutions
  - Quotient by colimits: $\mathsf{T}_{\Sigma_1+\Sigma_2}X = colimit\ \ \mathcal{D}_X$
  - Solve a simpler problem $S + F^\mu = S(FS)^\mu$

- **Definition:** A property P is modular for an operation $\oplus$ iff

$$\theta_1 \oplus \theta_2 \models P \text{ iff } \theta_1 \models P \wedge \theta_2 \models P$$

- **Semantics:** Model rewrite systems Θ as monads $T_\Theta : \mathbf{Pre} \longrightarrow \mathbf{Pre}$

- **Pre-requisite:** We need monadic formulation of properties

$$T \models P \quad \text{iff} \quad (X \models P \text{ implies } TX \models P)$$

- **Examples:** Works for CR and SN!

$$T_\Theta \models P \quad \text{iff} \quad \Theta \models P$$

- **Step 1:** Prove the monadic semantics is compositional

$$T_{\Theta_1+\Theta_2} = T_{\Theta_1} + T_{\Theta_2}$$

- **Step 2:** Express monad representing the structured TRS as a pointwise colimit

$$T_{\Theta_1+\Theta_2}(X) = \texttt{Colim}\ \mathcal{D}_X$$

- **Step 3:** Prove that if the objects of $\mathcal{D}_X$ satisfy the desired property, then so does $\texttt{Colim}\mathcal{D}_X$.

- **Claim:** The rewriting community were impressed

- **Motivation:** Consider adding substitutions to infinite terms
  - If we just add infinite substitutions, not all terms denote
  - Infinite terms are a monad: $F^\nu X = \nu Y.X + FY$

- **Answer 1:** Given signatures Σ, Ω want infinite $\Sigma + \Omega$ terms st
  - If we start at the root of the term and select any path
  - We may see an infinite segment of Σ-constructors
  - We may not see an infinite segment of Ω-constructors

- **Answer 2:** We want a universal property to reason with

$$F^\nu_\Sigma + F^\mu_\Omega$$

- **Motivation:** Proof theory is not modular
  - Consider cut elimination for $\lambda^{1,\times,\to}$. The proof is clearly modular but there are no modular proofs

- **Semantics:** Simple type theories are monads on **Gph**

$$\mathsf{T}_{\lambda^{1,\times,\to}} = \mathsf{T}_{\lambda^{\to}} + \mathsf{T}_{\lambda^{1,\times}}$$

- **Property:** A monad T satisfies cut-elimination iff there is a natural transform

$$\sigma : T \circ \mathsf{Pa} \longrightarrow \mathsf{Pa} \circ T$$

- **Theorem:** If T and S satisfy cut, so does $T + S$

Conclusions

- **Philosophy:** Category theory is rather beautiful
  - A few simple definitions and tools
  - Meta-theoretic intuitions to guide you
- **Practicalities:** Wide applications in mathematics and CS
  - Denotational semantics has been colonised
  - Other main applications are specification, rewriting, AI
- **Thanks!** I hope you enjoyed the course. I certainly did
  - Thanks to the other speakers for speaking in English
  - And ... they were pretty good.