Lambdas And Cartesian Closed Categories

A Tribute To Joachim Lambek
5 December 1922 – 23 June 2014

I’m Erik and I am addicted to reading language specifications.

I tried them all ....

C#

Visual
Basic

ECMA
Script

Hack

Swift

Dart

But there is one that is
even too strong for me.

Java 8 Lambdas/Method References

A method reference expression is compatible in an assignment context, invocation context, or casting context with a target type T if T is a functional interface type (§9.8) and the e
is congruent with the function type of the ground target type derived from T.

The ground target type is derived from T as follows:

If T is a wildcard-parameterized functional interface type, then the ground target typ
non-wildcard parameterization (§9.9) of T.

Otherwise, the ground target type is T.

A method reference expression is congruent with a function type if both of the following are true:

The function type identifies a single compile-time declaration corresponding to the reference.

One of the following is true:

The result of the function type is void.

The result of the function type is R, and the result of applying capture conversion (§5.1.10) to the return type of the invocation type (§15.12.2.6) of the chosen compile-time declaration is R' (where R is the target type that may be used to infer R'), and neither R nor R' is void, and R' is compatible with R in an assignment context.

You have not read the new PHP language specification yet?

https://thestrangeloop.com/sessions/project-lambda-in-java-8

Saint Leslie Lamport
Turing Award Winner 2014

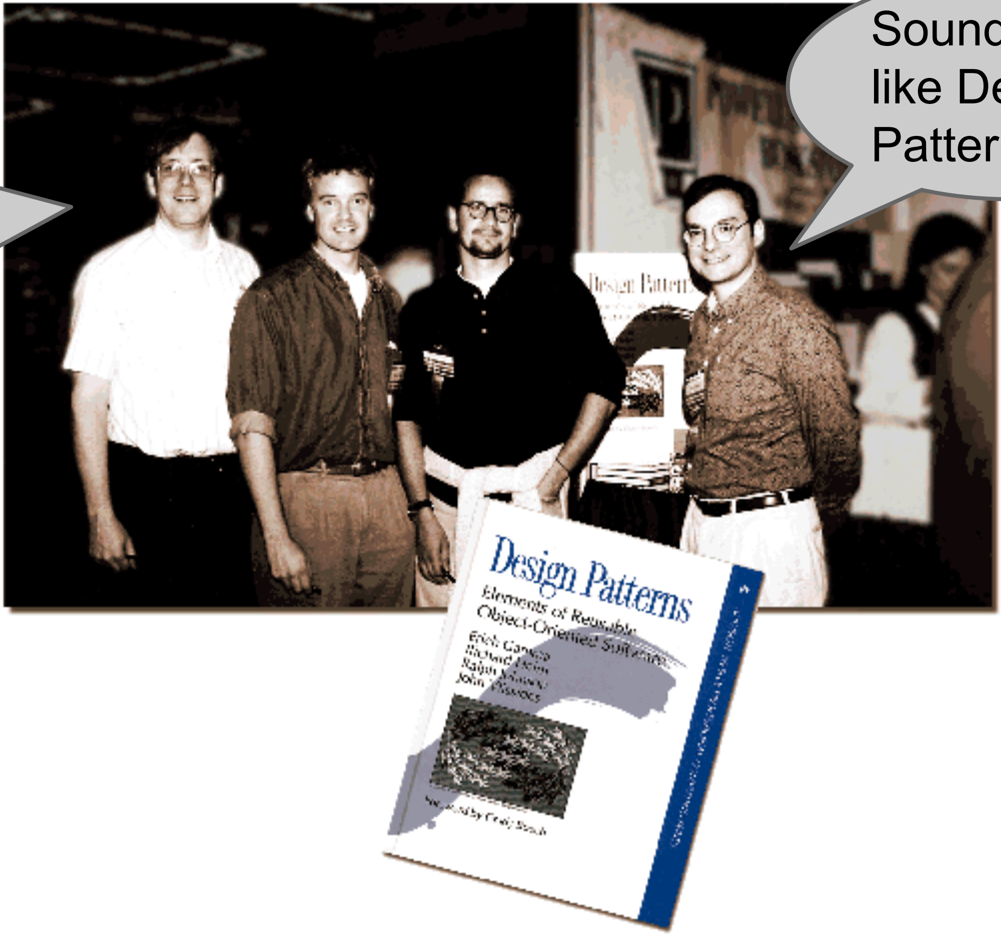
http://www.mariowiki.com/File:Crystalcoconut.gif

Abstract nonsense

From Wikipedia, the free encyclopedia

In mathematics, **abstract nonsense**, **general abstract nonsense**, and **general nonsense** are terms used facetiously by some mathematicians to describe certain kinds of arguments and methods related to category theory. (Very) roughly speaking, category theory is the study of the general form of mathematical theories, without regard to their content. As a result, a proof that relies on category theoretic ideas often seems slightly out of context to those who are not used to such abstraction, sometimes to the extent that it resembles a comical non sequitur. Such proofs are sometimes dubbed “abstract nonsense” as a light-hearted way of alerting people to their abstract nature.

“category theory is the study of the general form of mathematical theories, without regard to their content”



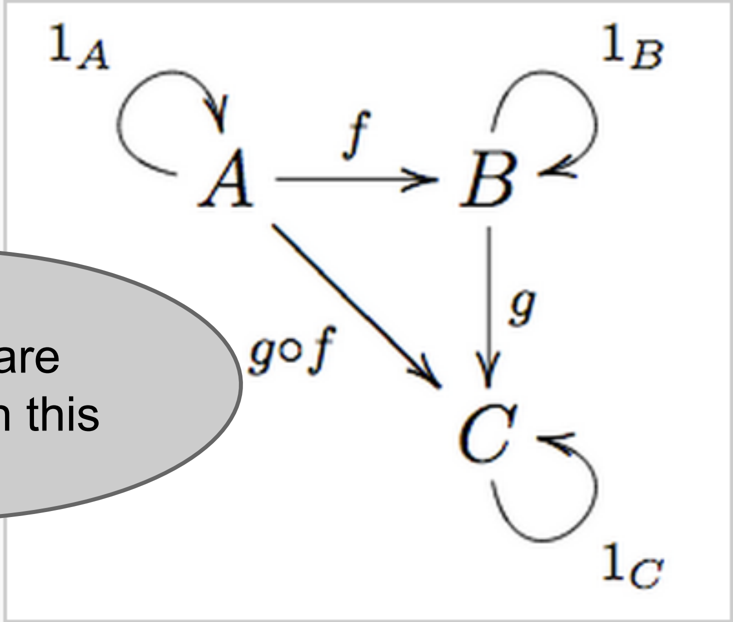
Sounds just like Design Patterns!

Category = Programming Language

Object = Type

Morphism = Static method f(a:A): B or property f: B on A

This is the problem we are going to fix in this talk.



This is a category with a collection of objects A, B, C and collection of morphisms denoted f, g, g ∘ f, and the loops are the identity arrows. This category is typically denoted by a boldface 3.

Category Theory
== Interface-based Modelling

Definition [edit]

Let $\mathcal{C}$ be a category with some objects X_1 and X_2. An object X is a product of X_1 and X_2, denoted $X_1 \times X_2$, iff it satisfies this universal property:

> there exist morphisms $\pi_1 : X \to X_1, \pi_2 : X \to X_2$ such that for every object Y and pair of morphisms $f_1 : Y \to X_1, f_2 : Y \to X_2$ there exists a unique morphism $f : Y \to X$ such that the following diagram commutes:

$$\begin{tikzcd}
 & Y \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\
X_1 & X_1 \times X_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & X_2
\end{tikzcd}$$

The unique morphism f is called the **product of morphisms** f_1 and f_2 and is denoted $\langle f_1, f_2 \rangle$. The morphisms π_1 and π_2 are called the **canonical projections** or **projection morphisms**.

Above we defined the **binary product**. Instead of two objects we can take an arbitrary family of objects indexed by some set I. Then we obtain the definition of a **product**.

An object X is the product of a family $\{X\}_i$ of objects iff there exist morphisms $\pi_i : X \to X_i$, such that for every object Y and a I-indexed family of morphisms $f_i : Y \to X_i$ there exists a unique morphism $f : Y \to X$ such that the following diagrams commute for all $i \in I$:

$$\begin{tikzcd}
 & X \arrow[d, "\pi_i"] \\
Y \arrow[ur, "f"] \arrow[r, "f_i"'] & X_i
\end{tikzcd}$$

The product is denoted $\prod_{i \in I} X_i$; if $I = \{1, \ldots, n\}$, then denoted $X_1 \times \cdots \times X_n$ and the product of morphisms is denoted $\langle f_1, \ldots, f_n \rangle$.

Let's Decode That Greek

Let $\mathcal{C}$ be a category with some objects X_1 and X_2.

Let $\mathcal{C}$ be a programming language with some types A and B.

And this noise

An object X is a product of X_1 and X_2, denoted $X_1{\times}X_2$, iff it satisfies this universal property: there exist morphisms $\pi_1 : X{\Rightarrow}X_1$, $\pi_2 : X{\Rightarrow}X_2$ such that for every object Y and pair of morphisms $f_1 : Y{\Rightarrow}X_1$, $f_2 : Y{\Rightarrow}X_2$ there exists a unique morphism $f_1{\triangle}f_2 : Y{\Rightarrow}X$ such that the following diagram commutes:

Commuting Diagram

$$\begin{tikzcd}
& Y \arrow[dl, "f_1"] \arrow[d, dashed, "f_1 \triangle f_2"] \arrow[dr, "f_2"'] & \\
X_1 & X_1 \times X_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & X_2
\end{tikzcd}$$

Translate to Equations

$$h = f_1 \triangle f_2$$

$$\Leftrightarrow$$

$$\pi_1 \circ h = f_1 \;\; \&\& \;\; \pi_2 \circ h = f_2$$

Is simply this specification

A type (A,B) is a product of A and B, iff it satisfies this universal property: there exist properties $\_1 : A$, $\_2 : B$ on (A,B) such that for every pair of methods $f(c\colon C)\colon A$, $g(c\colon C)\colon B$ there *exists* a factory method $f \triangle g(c\colon C)\colon (A,B)$ such that the following diagram commutes:

Commuting Diagram

$$
\begin{tikzcd}
& C \arrow[dl, "f"'] \arrow[d, dashed, "f \triangle g"] \arrow[dr, "g"] & \\
A & (A,B) \arrow[l, "\_1"] \arrow[r, "\_2"'] & B
\end{tikzcd}
$$

Translate to Equations

$$h = f \bigtriangleup g$$

$$\Leftrightarrow$$

$$h(c).\_1 = c.f$$

$$h(c).\_2 = c.g$$

Category Theory is Just
Interface-Based Design
With a Little Bit of
Additional Rigor

Scala Products

```
trait Product2[+T1, +T2] extends Product {
  abstract def _1: T1
  abstract def _2: T2
}

object Product2 {
  abstract def (f: C⇒A△g: C⇒B)(c: C): Product2[A,B]
}
```

Wishful thinking :->

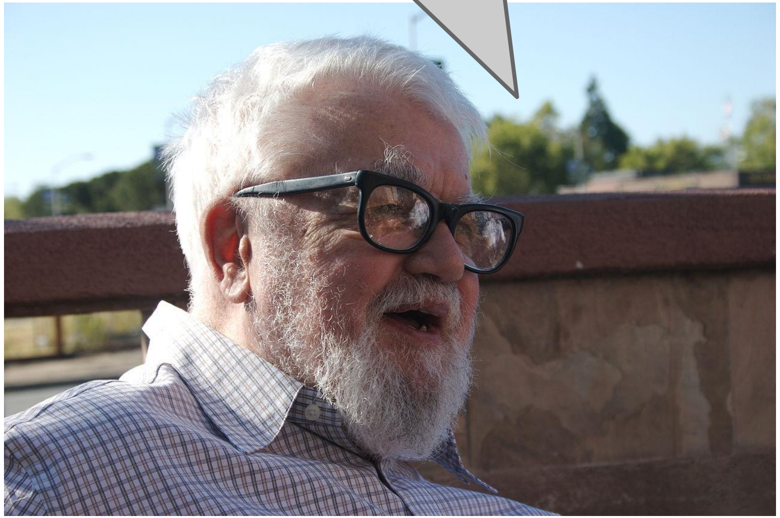
“There exists” is all we have

Given any two methods $f(c: C): A$ and $g(c:C):B$, we can define a new method $f \triangle g(c: C): (A,B) = (f(c), g(c))$.

Derived Functions, same deal

Given any two methods $f(a{:}\ A){:}C$ and $g(c{:}B){:}D$, we can define a new method $f{\times}g(ab{:}\ (A,B)){:}\ (C,D) = (f(ab.\_1), g(ab.\_2))$.

I could define △ and × generically in 1960 ...



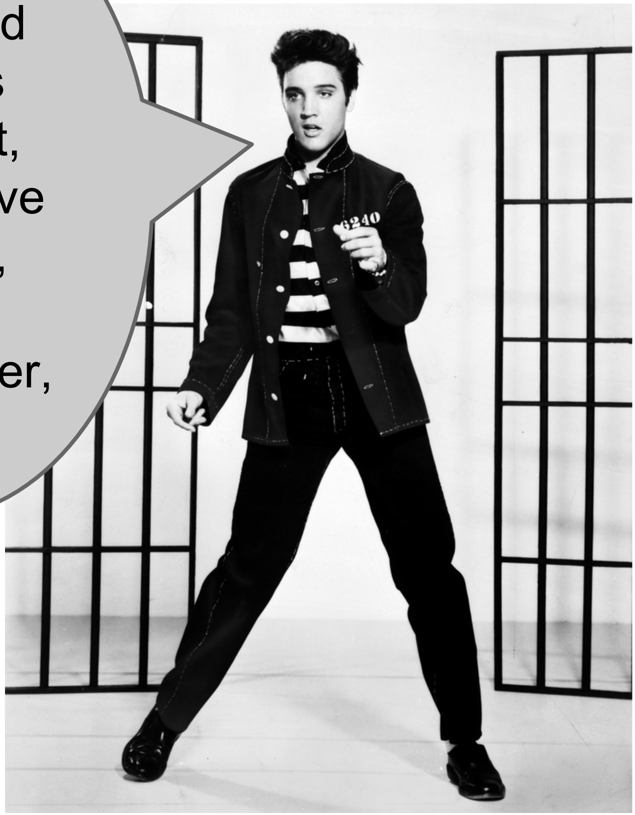
http://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)#mediaviewer/File:John_McCarthy_Stanford.jpg

Ha, ha, ha, I did it already in 1928.

http://www.learn-math.info/history/photos/Church.jpeg

We don’t need no stinkin’ delegates, we already have virtual methods

Don’t really understand any of this theory shit, but we have delegates, which are much better, of course.



https://netbeans.org/images_www/articles/73/javaee/ecommerce/intro/duke.png

http://upload.wikimedia.org/wikipedia/commons/9/99/Elvis_Presley_promoting_Jailhouse_Rock.jpg

Objects	Represent Real World Objects
Categories	Represent Mathematical Objects

Define not just interface, but also algebraic properties required of implementation

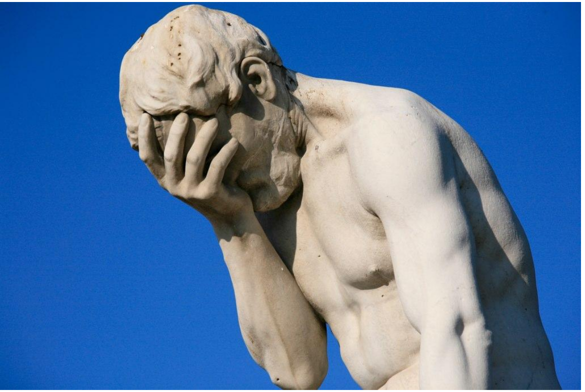
And don’t bullshit around with grandiloquent terms

Reality: A Cousin twice Removed

http://www.computer.org/portal/image/image_gallery?uuid=b15467b5-5dfe-44d3-8663-9e6cf406e8e4&groupId=141571&t=1270643684821

I thought this talk was about Cartesian Closed Categories. Cut the crap please!

Young man, before you dive into the deep end, let me note that in Smalltalk we had blocks since 1970.



http://en.wikipedia.org/wiki/Facepalm

Exponentials

Definition [edit]

Let C be a category with binary products and let Y and Z be objects of C. The exponential object Z^Y can be defined as a universal morphism from the functor $-\times Y$ to Z. (The functor $-\times Y$ from C to C maps objects X to $X\times Y$ and morphisms ϕ to $\phi\times \mathrm{id}_Y$).

Explicitly, the definition is as follows. An object Z^Y, together with a morphism

$$\mathrm{eval}\colon (Z^Y \times Y) \to Z$$

is an exponential object if for any object X and morphism $g : (X\times Y) \to Z$ there is a unique morphism

$$\lambda g\colon X \to Z^Y$$

such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[d, dashed, "\lambda g"'] & X \times Y \arrow[d, dashed, "\lambda g \times \mathrm{id}_Y"'] \arrow[dr, "g"] & \\
Z^Y & Z^Y \times Y \arrow[r, "\mathrm{eval}"'] & Z
\end{tikzcd}$$

If the exponential object Z^Y exists for all objects Z in C, then the functor that sends Z to Z^Y is a right adjoint to the functor $-\times Y$. In this case we have a natural bijection between the hom-sets

$$\mathrm{Hom}(X \times Y, Z) \cong \mathrm{Hom}(X, Z^Y).$$

(Note: In functional programming languages, the morphism *eval* is often called *apply*, and the syntax λg is often written *curry*(g). The morphism *eval* here must not to be confused with the eval function in some programming languages, which evaluates quoted expressions.)

The morphisms g and λg are sometimes said to be *exponential adjoints* of one another.[1]

Lets decode the Greek

Let $\mathcal{C}$ be a category with binary products and let Y and Z be objects of $\mathcal{C}$. The exponential object Z^Y can be defined as a universal morphism from the functor $-\times Y$ to Z. (The functor $-\times Y$ from $\mathcal{C}$ to $\mathcal{C}$ maps objects X to $X\times Y$ and morphisms φ to $\varphi\times id$).

Lets decode the Greek

Let L be a language that supports tuples and let A and B be types of L. The function type $A \Rightarrow B$ can be defined as a factory method from the functor $- \times A$ to B. (The functor $- \times A$ in L maps types C to $C \times A$ and methods m to $m \times id$).

What Is A Functor?

Definition [edit]

Let *C* and *D* be categories. A **functor** *F* from *C* to *D* is a mapping that[3]

- associates to each object $X \in C$ an object $F(X) \in D$,
- associates to each morphism $f : X \to Y \in C$ a morphism $F(f) : F(X) \to F(Y) \in D$ such that the following two conditions hold:
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X \in C$
  - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$.

That is, functors must preserve identity morphisms and composition of morphisms.

Covariance and contravariance [edit]

There are many constructions in mathematics that would be functors but for the fact that they "turn morphisms around" and "reverse composition". We then define a **contravariant functor** *F* from *C* to *D* as a mapping that

- associates to each object $X \in C$ an object $F(X) \in D$,
- associates to each morphism $f : X \to Y \in C$ a morphism $F(f) : F(Y) \to F(X) \in D$ such that
  - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object $X \in C$,
  - $F(g \circ f) = F(f) \circ F(g)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$.

Note that contravariant functors reverse the direction of composition.

Ordinary functors are also called **covariant functors** in order to distinguish them from contravariant ones. Note that one can also define a contravariant functor as a *covariant* functor on the opposite category C^{op}.[4] Some authors prefer to write all expressions covariantly. That is, instead of saying $F : C \to D$ is a contravariant functor, they simply write $F : C^{\mathrm{op}} \to D$ (or sometimes $F : C \to D^{\mathrm{op}}$) and call it a functor.

Contravariant functors are also occasionally called *cofunctors*.

Lets decode the Greek

Let C be a category. A functor F is a mapping that associates to each object X an object $F(X)$, and associates to each morphism $f{:}X \Rightarrow Y$ a morphism $F(f){:}F(X) \Rightarrow F(Y)$ such that the following two conditions hold: $F(id) = id$ and $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f{:}X \Rightarrow Y$, and $g{:}Y \Rightarrow Z$. That is, functors must preserve identity morphisms and composition of morphisms.

Lets decode the Greek

Let L be a language. A functor $C[\_]$ is a generic type that associates to each type A an instantiated type $C[A]$, and has a method $map(f: A \Rightarrow B): C[B]$ such that for all $cs: C[A]$ the following two conditions hold: $cs.map(id) = cs$ and $cs.map(a \Rightarrow f(g(a)) = cs.map(g).map(f)$ for all functions $f: B \Rightarrow C$, and $g: A \Rightarrow B$. That is, functors must preserve identity and composition of functions.

No worries, I am a dirty hacker. Purity is sooooo overrated. A function is a type that reifies methods.

Doh!

That’s what exponentials are for.

You have messed up a bit by silently introducing functions already, which is the type we were trying to define ….

http://andrej.com/mathematicians/large/Lambek_Joachim.jpg

Continue to Decode the Greek

Explicitly, the definition is as follows. An object Z^Y, together with a morphism $eval$: $(Z^Y \times Y) \Rightarrow Z$ is an exponential object if for any object X and morphism g: $(X \times Y) \Rightarrow Z$ there is a unique morphism λg: $X \Rightarrow Z^Y$ such that the following diagram commutes:

Commuting Diagram

$$\begin{tikzcd}
X \arrow[d, dashed, "\lambda g"'] & X \times Y \arrow[d, dashed, "\lambda g \times id"'] \arrow[dr, "g"] & \\
Z^Y & Z^Y \times Y \arrow[r, "eval"'] & Z
\end{tikzcd}$$

Translate to Equations

To be precise Erik, Λg is the *unique* morphism that makes this equation hold

$$g(a,b)$$
$$=$$
$$(eval \circ \Lambda g \times id)(a,b)$$
$$=$$
$$eval\ (\Lambda g(a),\ b)$$

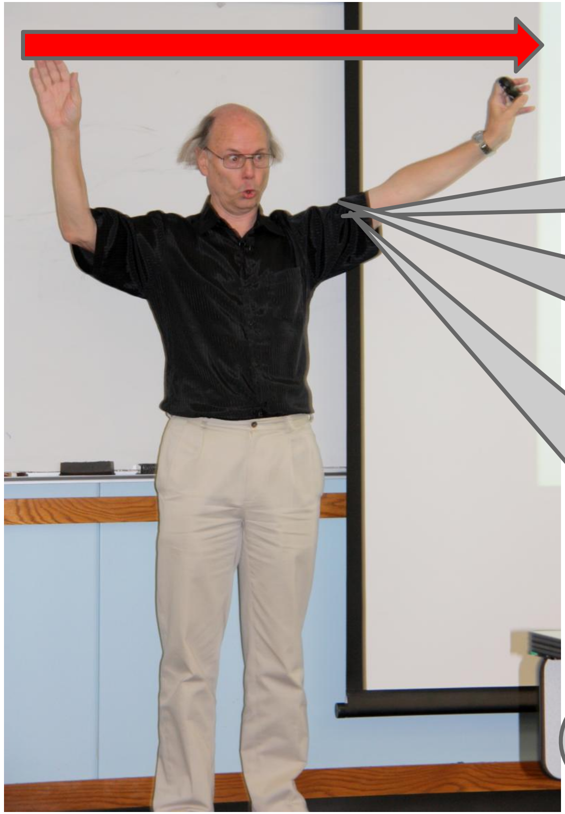
You may also recognize λg as "currying" in Haskell, i. e.

curry g a b = g(a,b)
uncurry f(a,b) = f a b

Which I invented in 1958.

http://www.haskell.org/wikiupload/8/86/HaskellBCurry.

Products and Objects



As I explained in 1979, an instance method is just a static method that takes the this pointer as an additional argument

Hence an instance method m (b:B): C on type A is simply a morphism from $(A \times B) \Rightarrow C$.

And did I already mention that C++11 has lambdas?

http://www.cse.unt.edu/site/node/517

This is argument 0 to a method call

The Java Virtual Machine uses local variables to pass parameters on method invocation. On class method invocation, any parameters are passed in consecutive local variables starting from local variable *0*. On instance method invocation, local variable *0* is always used to pass a reference to the object on which the instance method is being invoked (this in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable *1*.

Continue to Decode the Greek

Explicitly, the definition is as follows. A type $B \Rightarrow C$, together with a method $apply$ (b: B): C is a *function type* if for any type A and method m(b: B):C on A there is a factory method (a: A)::m :B$\Rightarrow$$C$ such that (C)a::m.apply(b)=(C)a.m(b).

Translate to Equations

a::m.apply(b)

=

a.m(b)

If the exponential object Z^Y exists for all objects Z in C, then the functor that sends Z to Z^Y is a right adjoint to the functor $- \times Y$. In this case we have a natural bijection between the hom-sets

$$\mathrm{Hom}(X \times Y, Z) \cong \mathrm{Hom}(X, Z^Y).$$

Hom-set?

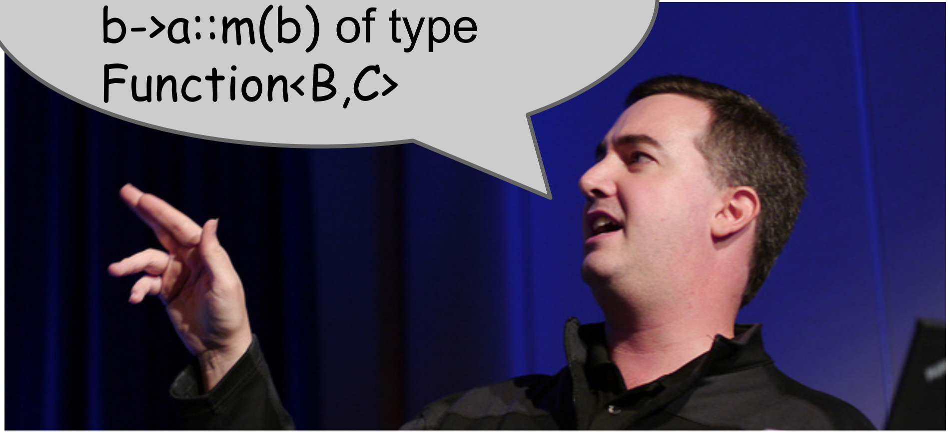
Just a fancy way to say “all methods m (a: A): B”

$$\mathrm{Hom}(A,B) = \{ m \mid m \in C \;\&\; m\colon A \Rightarrow B \}$$

The Magic Of Method References

$$\mathrm{Hom}(A\times B,C) \cong \mathrm{Hom}(A,B\Rightarrow C)$$

Instance methods m(b: B):C on an instance a: A



Are isomorphic to lambda expressions b->a::m(b) of type Function<B,C>

Unfinished Functor Business

In the most concise symmetric definition, an adjunction between categories *C* and *D* is a pair of functors,

$$F : \mathcal{D} \to \mathcal{C} \quad \text{and} \quad G : \mathcal{C} \to \mathcal{D}$$

and a family of bijections

$$\hom_{\mathcal{C}}(FY, X) \cong \hom_{\mathcal{D}}(Y, GX)$$

which is natural in the variables *X* and *Y*. The functor *F* is called a **left adjoint functor**, while *G* is called a **right adjoint functor**. The relationship "*F* is left adjoint to *G*" (or equivalently, "*G* is right adjoint to *F*") is sometimes written

$$F \dashv G.$$

In our case

$$\mathrm{Hom}(\mathbf{A}\times_B,\mathbf{C}) \cong \mathrm{Hom}(\mathbf{A},_B\Rightarrow\mathbf{C})$$

$$\mathrm{Hom}(F[_B,\mathbf{A}],\mathbf{C}) \cong \mathrm{Hom}(\mathbf{A},G[_B,\mathbf{C}])$$

Show Me Some Code

```
object F {
  def apply[A,B](a: A, b: B): F[B,A] = new F(a,b)
}

class F[B,A](_1 : A, _2 : B) extends Pair[A,B](_1, _2) {
  def map[C](f: A=>C): F[B,C] = F(f(_1),_2)
}
```

Show Me Some Code

```
object G {
  def apply[B,C](f: B=>C): G[B,C] = new G[B,C] {
    override def apply(b: B): C = f(b)
  }
}

trait G[B,C] extends (B=>C) {
  def map[C](f: C=>A): G[B,A] = G(b => f(apply(b)))
}
```

Homework

Show that `F[B,_]` and `G[B,_]` are indeed *functors*.

Show that `F[B,_]` and `G[B,_]` are *adjoint functors* by defining implicit conversions each direction.

Monads And Adjunctions

Monads [edit]

Every adjunction ⟨*F*, *G*, ε, η⟩ gives rise to an associated monad ⟨*T*, η, μ⟩ in the category *D*. The functor

$$T : \mathcal{D} \to \mathcal{D}$$

is given by *T* = *GF*. The unit of the monad

$$\eta : 1_{\mathcal{D}} \to T$$

is just the unit η of the adjunction and the multiplication transformation

$$\mu : T^2 \to T$$

is given by μ = *G*ε*F*. Dually, the triple ⟨*FG*, ε, *F*η*G*⟩ defines a comonad in *C*.

Every monad arises from some adjunction—in fact, typically from many adjunctions—in the above fashion. Two constructions, called the category of Eilenberg–Moore algebras and the Kleisli category are two extremal solutions to the problem of constructing an adjunction that gives rise to a given monad.

The monad that arises from F[B,_] and G[B,_] is the State Monad

But that is a topic for another talk

http://www.cs.kent.ac.uk/people/staff/rej/face.gif

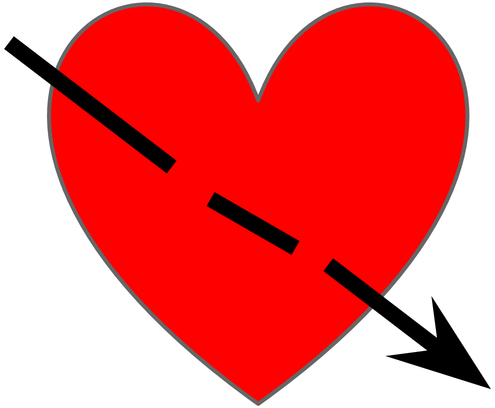
Your Next Tattoo

eval(λm(a), b)

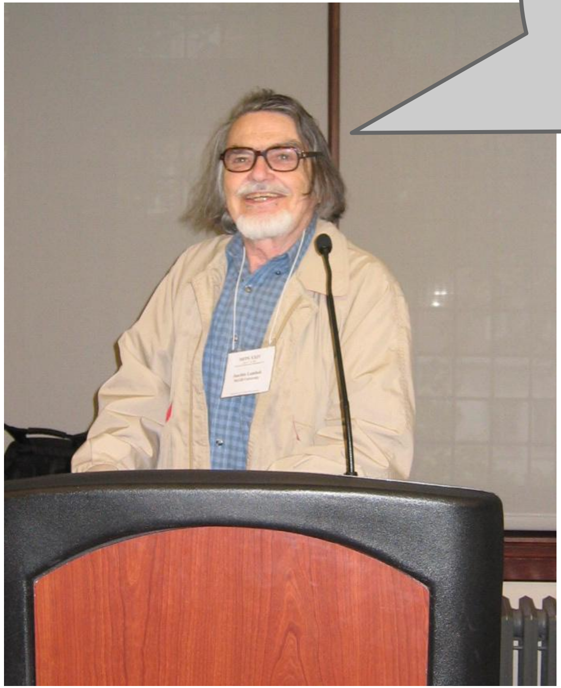
a::m.apply(b)

http://en.wikipedia.org/wiki/Jolly_Roger

Method References



Are Exponentials



What keeps puzzling me is why did it take so damn long before OO programmers made methods into first class objects.

Because I distracted them with Spring, and got stinking rich doing it ;-)

http://blogs.vmware.com/tp/.
a/6a00d8341c328153ef0120a4e05
530970b-pi