# Easy Categories For Programmers

## With Java Implementation

**Vlad Patryshev**

**1/21/2009**

# Table of Contents

# I. Prior Art

## 1. Backgrounds

I'm going to talk about categories. Categories in general are not defined in terms of sets; categories that are sets are called "small categories".
But for the sake of simplicity I'm going to use sets, in most cases. More, the code examples will use finite enumerable sets. Everybody knows, or thinks that knows, what "finite" means; "enumerable" means (assuming boolean logic) that there is a recursive algorithm generating all the elements of the set.

This is the main difference between sets defined in libraries like in Java or Scala and sets in general: general sets, e.g. those defined by Zermelo axioms, can be expressed in code as predicates; since there is no reasonable way to list all objects that satisfy a given predicate, all calculations become either impossible or extremely inefficient. We should not worry about it, though. Quoting my colleague OD, *"For an average coder a 'set' is a kind of a data structure, not fancy stuff Zermelo wrote about."* (See more of his comments here.)

Also, we have to remember that maintaining set-theoretical integrity in Java or Scala library sets is not a realistic task. Look at this piece of Java code:

```
Set<Set<?>> ss1 = new HashSet<Set<?>>();
Set<Set<?>> ss2 = new HashSet<Set<?>>();
Set<Integer> si = new HashSet<Integer>() {{ add(0); }};
ss1.add(si);
ss1.add(ss1);
ss1.add(ss2);
ss2.add(si);
ss2.add(ss1);
System.out.println(ss1);
```

It'll produce stack overflow; this would not happen if we had a true set theory (e.g. ZF); but checking this takes too much time. We don't have that much time. (Actually we do, but if we admit it, then our competitors win.)

Another question is, how do we represent mappings from one set to another. We can have a function type with given domain (where the function as defined) and codomain (where the function takes values) types; or we can use `Map<From, To>`, or we could be very set-theoretical and use sets of pairs satisfying certain conditions. The main motivation in choosing one over the other is convenience; it is not apriori obvious which approach is more convenient; we probably will have to combine all three.

Java lacks some helpful classes; I'll be adding them here when needed.

### 1.1. Note for Programmers

Why would a programmer be interested in categories? Well, without categories, Haskell and all that new stuff is more or less incomprehensible. But I hope to go ahead way beyond, to toposes and topos logic. To learn all this, you'll need to have categories behind your belt as a pretty simple and convenient tool for representing objects and their relationships.

The source code is stored here: http://myjavatools.com/projects/Category/source.zip. There's actually

more code in the archive than what is published here. Let me know if you have problems working with the code.

Don't forget to set '-ea' jvm parameter, to enable asserts.

## 1.2. Note for Mathematicians

You probably won't find anything new here. Note that while I use the term "set", I actually do not rely on any particular set theory or computational theory; the code here is not expected to run on Turing machines; more, if you think Turing, you should rather think Heyting. It's a different universe. For instance, a set (or an object) A is *finite* if any subset of A that is isomorphic to A equals A. Why do I say "object", not "set"? Because sets are not needed here; it's enough if it is discrete objects in a topos. Being discrete is not needed too, but the users will get scared to death if we start offering them half-units or double rings that do not satisfy the AC even for two elements (for which I have to thank professor Andre Scedrov).

## 1.3. Note for Readers

I am very, very thankful to all the good people that read it and make comments (let me know, guys, if you want your "real" names disclosed): OD, warrior, smalgin, igor sereda, Кристофер Робин, Eugene Kirpichov, nivanych, 109, faceted_jacinth...

There are many sources explaining category stuff in details. This article is good, although not always correct; I would rather recommend the classical "Categories for the Working Mathematician" by S. MacLane.

## 1.4. Java Code

Here are some classes we will need later on:

```
package math.cat;

import java.util.*;


/**
 * Base tools used in this package.
 *
 * All code is <a href="http://myjavatools.com/projects/Category/source.zip">here</a>
 *
 * Credits: http://antilamer.livejournal.com/245962.html
 */

public class Base {
  public static <T> Set<T> Set(T... elements) {
    return new HashSet<T>(Arrays.asList(elements));
  }

  public static <T> T[] array(T... elements) {
    return elements;
  }

  public static <K,V> Map<K,V> Map(final K[] keys, final V[] values) {
    assert keys.length == values.length;
    Map<K,V> map = new HashMap<K,V>();
    for (int i = 0; i < keys.length; i++) {
      map.put(keys[i], values[i]);
    }
    return Collections.unmodifiableMap(map);
  }
```

```
  /**
   * Extracts a set from a string as produced by Set.toString().
   * Entries in the set cannot be empty strings and cannot contain commas.
   *
   * @param s the string
   * @return restored set
   */
  public static Set<String> parseSet(String s) {
    String[] content = s.substring(1, s.length() - 1).split(",\\s*");
    Set<String> result = new HashSet<String>();
    for (String entry : content) {
      String trimmed = entry.trim();
      if (!trimmed.isEmpty()) result.add(trimmed);
    }
    return result;
  }
}
```

## 2. PoSet

Before categories, let's play with simpler structures (but slightly more complicated than just sets); partial order (or poset, partially-ordered set) is the simplest. In Java there is a class named SortedSet; in such a class every elements are comparable. Not so in partial order.

### 2.1. Definition

Partially-ordered set is a set and a binary relationship that is reflexive, transitive and antisymmetric. That is, we have three rules:

- `x <= x`
- `if x <= y && y <= z, then x <= z`
- `if x <= y && y <= x, then x == y`

### 2.2. Examples

Any set `X` can be considered partially-ordered: no order at all is a partial order.
Natural numbers, where order is defined as this: `a <= b` if `b` is divisible by `a`.
If `(X, <=)` is a partial order, the opposite, `(X, >=)` is also a partial order.

### 2.3. Java Code

```
package math.cat;

import java.util.*;

import static math.cat.Pair.*;
import math.cat.Pair;

/**
 * Sample Implementation of partially ordered set.
 * All code is <a href="http://myjavatools.com/projects/Category/source.zip">here</a>
 */
public abstract class PoSet<T> extends AbstractSet<T> {
  private final Set<T> elements;

  /*
   * Java technicalities: have to override these methods.
   */
  @Override public Iterator<T> iterator() {  return elements.iterator(); }
```

```
  @Override public int size() { return elements.size(); }
  @Override public int hashCode() { return elements.hashCode(); }
  @Override public boolean equals(Object o) { return o instanceof PoSet &&
((PoSet)o).equals(this); }

  /**
   * Defines partial order.
   * @param x first compared element
   * @param y second compared element
   * @return true if x is before y in this partial order
   */
  public abstract boolean _le_(T x, T y);

  /**
   * Basic constructor. You need to define a comparator method to build an instance.
   * @param elements elements of this poset.
   */
  public PoSet(Set<T> elements) {
    this.elements = elements;
    validate();
  }

  /**
   * Validates the axioms of this partially ordered set
   */
  private void validate() {
    for(T x : elements) {
      assert _le_(x, x): " reflexivity broken at " + x;

      for (T y : elements) {
        if (_le_(x, y)) {
          if (_le_(y, x)) assert x.equals(y) : "antisymmetry broken at "+ x +", "+ y;

          for (T z : elements) {
            if (_le_(y, z)) assert _le_(x, z) : "transitivity broken at "+ x +", "+ y +", "+ z;
          }
        }
      }
    }
  }

  /**
   * Two posets are equal if they have the same elements and partial order is the same.
   *
   * @param other other poset to compare
   * @return true if these two posets are equal
   */
  private boolean equals(PoSet<T> other) {
    boolean isEqual = elements.equals(other.elements);
    for (T a : elements) for (T b : elements) {
      isEqual = isEqual && (_le_(a, b) == other._le_(a, b));
    }
    return isEqual;
  }
}
```

Here's how we could build a poset:

```
 PoSet<String> ex1 = new PoSet<String>(Set("abc", "def", "ab", "defgh")) {
  public boolean _le_(String a, String  b) {
   return b.indexOf(a) >= 0;
  }
 };
```

# 3. Graphs

## 3.1. Definition

There are many different flavors of graphs. Here we deal with what some people call directed multigraph: we are given a set of nodes `N` and a set of edges (or arrows) `A`; each edge (arrow) originates at a node and ends at a node. An arrow (edge) may end at the same node where it originated, but not necessarily; there can be any number of arrows leading from one node to another.

Seems like the mainstream notion of graph is when there's not more than one arrow from one node to another; and quite often "graph" means undirected graph, that is, a set of nodes `N` and a symmetric non-reflexive binary relationship `R ⊂ N×N` that defines which nodes are connected. A ~~pedestrian~~ driver example is San Francisco: an undirected graph consists of two-way streets, and a directed graph consists of one-way streets.

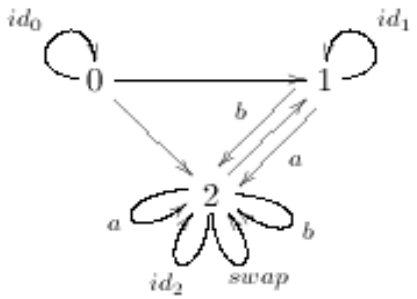
From categorical point of view, though, directed multigraphs make more sense.

I use an unusual (for a graph-theorist) notation: source and target of an arrow are returned by functions called `d0` and `d1`. In category theory source is called *domain*, and target is called *codomain*. The notation, `d0` and `d1`, comes from so-called *internal categories* (we'll touch them later on) - that's my excuse for using confusing terms.

## 3.2. Example

Take three sets: an empty set **0**, a singleton **1**, and a two-element set **2** (consisting of elements named **a** and **b**). How many mappings can we find between these three?

Let's start with **0**, an empty set. An empty set is a subset of any other, so there's just one function from **0** to each of the three, the inclusion. Also, there could be no function from a non-empty map to an empty one.
Then take **1**. A function from a single-element set to another set represents an element of that set; so we have an identity map **id**:**1**->**1**, and two maps to the two-element set **2**, corresponding to its elements, **a** and **b**.
Now take **2.** It has a trivial function (**2.1**) to **1**. Then what are all possible functions from a two-element set to itself? **id**, an identity, then **swap**, the one that swaps **a** and **b**; and then there are two functions that maps both source elements to the same target element (**a** or **b**)




here **id** is identity map, **swap** swaps **a** and **b**. If we treat the three sets as nodes, and maps as arrows, we have a pretty generic example.
It is probably obvious that any set can be considered a graph with an empty set of arrows.

It is probably obvious that any set can be considered a graph with an empty set of arrows.

Another generic example is a poset treated as a graph: elements of the poset become nodes, and

arrows are pairs `(x,y)` such that `x<=y` in the poset.

## 3.3. Java Code

```
package math.cat;

import java.util.AbstractSet;
import java.util.Iterator;
import java.util.Set;
import java.util.Collections;

/**
 * Sample Implementation of (oriented multi-) graph.
 * All code is <a href="http://myjavatools.com/projects/Category/source.zip">here</a>
 */
public abstract class Graph<N, A> extends AbstractSet<N> {
  private Set<N> nodes;

  /*
   * Java technicalities: have to override these methods.
   */
  @Override public Iterator<N> iterator() {  return nodes.iterator(); }
  @Override public int size() { return nodes.size(); }
  @Override public int hashCode() { return nodes.hashCode() * 2 + arrows().hashCode(); }
  @Override public boolean equals(Object o) { return o instanceof Graph && equals((Graph)o); }

  /**
   * Lists all nodes of a graph.
   * @return a set of nodes
   */
  public Set<N> nodes() { return Collections.unmodifiableSet(nodes); }

  /**
   * Lists all arrows of a graph.
   * @return a set of arrows
   */
  public abstract Set<A> arrows();

  /**
   * Maps an arrow to its domain, aka source.
   * @param arrow an arrow
   * @return the arrow's domain (source)
   */
  public abstract N d0(A arrow);
  /**
   * Maps an arrow to its codomain, aka target.
   * @param arrow an arrow
   * @return the arrow's codomain (target)
   */
  public abstract N d1(A arrow);

  /**
   * An abstract graph constructor. Takes nodes; arrows are defined in subclasses.
   *
   * @param nodes graph nodes.
   */
  protected Graph(Set<N> nodes) {
    this.nodes = nodes;
    validate();
  }

  /**
   * Validates the graph: checks that for each arrow its domain and codomain is in the graph.
   */
  public void validate() {
    for (A arrow : arrows()) {
      assert nodes.contains(d0(arrow)) : "Domain for " + arrow + " not defined";
      assert nodes.contains(d1(arrow)) : "Codomain for " + arrow + " not defined";
```

```
    }
  }

  /**
   * Checks equality of this graph to that one.
   * They are equal if they have the same sets of nodes and arrows, and the arrows
   * originate and end at the same nodes.
   *
   * @param that another graph
   * @return true if they are equal.
   */
  private boolean equals(Graph<N,A> that) {
    boolean isEqual =
        this.nodes().equals(that.nodes()) && // same nodes?
        this.arrows().equals(that.arrows()); // same arrows?

    for (A arrow : arrows()) {
      isEqual = isEqual &&
          this.d0(arrow).equals(that.d0(arrow)) && // same domain?
          this.d1(arrow).equals(that.d1(arrow));   // same codomain?
    }
    return isEqual;
  }
}
```

This is a barebone code; we need to add a reasonable constructor or factory to build something meaningful.

So, for example, we can throw in this:

```
 /**
  * Builds a graph from given nodes and arrows.
  *
  * @param nodes graph nodes
  * @param arrows graph arrows, represented here as mapping arrow tags to (domain,codomain)
pairs.
  * @return a new graph
  */
 public static <N,E> Graph<N,E> Graph(Set<N> nodes, final Map<E, Pair<N,N>> arrows) {
  return new Graph<N,E>(nodes) {
   public N d0(E edge) { return arrows.get(edge).x(); }
   public N d1(E edge) { return arrows.get(edge).y(); }
   public Set<E> arrows() { return arrows.keySet(); }
  };
 }
```

Now we can build a graph from the picture above:

```
   Set<Integer> nodes = Set(0, 1, 2);
  Map<String, Pair<Integer, Integer>> arrows =
     Map(array(   "0.id",     "0.1",     "0.2",    "1.id",      "a",        "b",     "2.id",
"2.a",     "2.b", "2.swap"),
       array(Pair(0,0), Pair(0,1), Pair(0,2), Pair(1,1), Pair(1,2), Pair(1,2), Pair(2,2),
Pair(2,2), Pair(2,2), Pair(2,2)));
  Graph<Integer, String> three = Graph(nodes, arrows);
```

A decent `toString()` method (not shown here, see codebase) will produce the following output:

```
([0, 1, 2], {2.a: 2->2, 1.id: 1->1, 2.b: 2->2, 0.2: 2->0, b: 2->1, 0.1: 1->0, a: 2->1, 2.swap: 2-
>2, 2.id: 2->2, 0.id: 0->0})
```

And now a factory that build a graph out of a poset:

```
public static <N> Graph<N, Pair<N, N>> Graph(PoSet<N> poset) {
 final Set<Pair<N, N>> arrows = new HashSet<Pair<N, N>>();
 for (N x : poset) for (N y : poset) if (poset._le_(x, y)) {
  arrows.add(Pair(x, y));
 }
 return new Graph<N, Pair<N, N>>(poset) {
  public N d0(Pair<N, N> arrow) { return arrow.x(); }
  public N d1(Pair<N, N> arrow) { return arrow.y(); }
  public Set<Pair<N, N>> arrows() { return arrows; }
 };
}
```

# II. Meet Categories

## 4. Category

### 4.1. Definition

A *category* is a graph with two operations defined on objects and arrows:

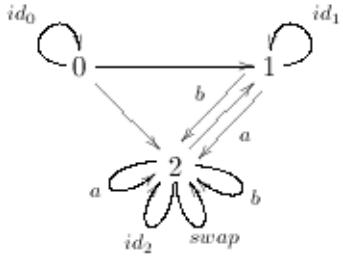
- For each two consecutive arrows `f` and `g` (consecutive means that `d1(f) = d0(g)`: the codomain of `f` is the domain of `g`) a *composition* arrow `h = g o f` is defined, such that `d0(h) = d0(f)` and `d1(h) = d1(g)`. In other words, for `f: a -> b, g: b -> c, h = g o f : a -> c`. Note that while the sequence of arrows is first `f` then `g`, the old tradition forces us to write the arrows in the opposite order ("in the order of their application, as if they were functions applied to an argument).
- For each object a there is a *unit* arrow $1_a$`: a -> a`, such that `f o` $1_a$ `= f` for each `f: a -> b`, and $1_a$ `o f = f` for each `f : b -> a`.

Composition is required to be associative: `(h o g) o f = h o (g o f)`.

People often tend to interpret arrows in categories as some kind of functions that map elements of their domains to elements of their codomains.

### 4.2. Example

Remember the graph example from 3? Let's redraw it here. It is convenient to draw categories as diagrams:




In this diagram **id** denotes unit arrows; composition is defined (in non-trivial cases) like this:

**a** o **a** = **a**; **2.b** o a = **b**; **2.a o b** = **a**; **2.b** o b= **b**; **swap** o a = **b**; **swap** o b = **a**; **swap** o **swap** = $\mathbf{1_2}$.

### 4.3. Java Implementation

```
package math.cat;

import java.util.Set;

/**
 * Sample Implementation of category.
 * All code is <a href="http://code/google.com/p/categories/">here</a>
 */
public abstract class Category<O, A> extends Graph<O, A> {

  public Set<O> objects() { return nodes(); }
  public abstract A unit(O x);
  public abstract A m(A f, A g); // produces a composition of f followed by g

  /*
   * Java technicalities: have to override these methods.
   */
  @Override public boolean equals(Object o) {
```

```
    return o instanceof Category && equals((Category)o);

  }

  private Category(Set<O> objects) {
    super(objects);
    validate();
  }

  /**
   * Validates this category, checking all the axioms.
   */
  public void validate() {
    super.validate();

    for (O x : objects()) {
      A unit = unit(x);
      assert arrows().contains(unit) : "Unit for " + x + " not defined";
      assert d0(unit).equals(x) : "Domain of unit " + unit + " should be " + x;
      assert d1(unit).equals(x) : "Codomain of unit " + unit + " should be " + x;
    }

    for (A f : arrows()) {
      assert m(unit(d0(f)), f).equals(f) : "Left unit law broken for "+ unit(d0(f)) +" and "+ f;
      assert m(f, unit(d1(f))).equals(f) : "Right unit law broken for "+ unit(d0(f)) +" and "+ f;
    }

    for (A f : arrows()) for (A g : arrows()) if (d1(f).equals(d0(g))) {
      A gf = m(f, g);
      assert gf != null : "Composition of " + f + " and " + g + " not defined";
      assert d0(gf).equals(d0(f)): "Wrong composition "+ gf +" of "+ f +" and "+ g +": its domain
is "+ d0(gf) +", must be "+ d0(f);
      assert d1(gf).equals(d1(g)): "Wrong composition "+ gf +" of "+ f +" and "+ g +": its
codomain is "+ d1(gf) +", must be "+ d1(g);
    }

    for (A f : arrows()) for (A g : arrows()) for (A h : arrows()) {

      if (d1(f).equals(d0(g)) && d1(g).equals(d0(h))) {
        assert m(m(f, g), h).equals(m(f, m(g, h))) : "Associativity broken for "+ f +", "+ g +"
and "+ h;

      }
    }
  }

  private boolean equals(Category<O, A> other) {
    boolean isEqual = // two categories are equal if:
        objects().equals(other.objects()) && // they have the same objects
        arrows().equals(other.arrows()); // and they have the same arrows

    for (O x : objects()) {
      isEqual = isEqual && (unit(x).equals(other.unit(x))); // objects have the same unit arrows
    }

    for (A f : arrows()) {
      isEqual = isEqual &&
          (d0(f).equals(other.d0(f))) && // and arrows have the same domains
          (d1(f).equals(other.d1(f)));   // and the same codomains
      for (A g : arrows()) if (d1(f).equals(d0(g))) {
        isEqual = isEqual && m(f, g).equals(other.m(f, g)); // and arrow composition is the same
      }
    }
    return isEqual;
  }
}
```

## 5. More Java

The code above was a minimal set of methods that ensure category properties; we have to add some factory methods. A simple factory method just fills in all the slots:

```
public static <O, A> Category<O, A> Category(
     final Set<O> objects,
     final Map<A, Pair<O, O>> arrows,
     final Map<O, A> units,
     final Map<Pair<A, A>, A> composition) {
   return new Category<O, A>(objects) {
     public O d0(A f) { return arrows.get(f).x(); }
     public O d1(A f) { return arrows.get(f).y(); }
     public A unit(O x) { return units.get(x); }
     public A m(A f, A g) { return composition.get(Pair(f, g)); }
     public Set<A> arrows() { return arrows.keySet(); }
   };
 }
```

This factory method is okay, it builds a category... but it requires too much information. We already know that units should have the same domain and codomain; we know that they are neutral in relation to composition; why bother to specify all this then?

So, we could make it smarter (sacrificing some generality):

```
public static <A> Category<A, A>
     buildCategory(final Set<A> units,
                   final Map<A, A> d0,
                   final Map<A, A> d1,
                   final Map<Pair<A, A>, A> mSource) {
   final Map<A, A> domain = new HashMap<A, A>(d0);
   final Map<A, A> codomain = new HashMap<A, A>(d1);
   final Map<Pair<A, A>, A> m = new HashMap<Pair<A, A>, A>(mSource);
   for(A unit : units) {
     domain.put(unit, unit); // define d0 for unit arrows
     codomain.put(unit, unit); // define d1 for unit arrows
     for (A f : domain.keySet()) { // define composition for unit arrows
       if (domain.get(f).equals(unit)) {
         m.put(Pair(unit, f), f);
       }
       if (codomain.get(f).equals(unit)) {
         m.put(Pair(f, unit), f);
       }
     }
   }

   fillCompositionTable(domain, codomain, m);

   return new Category<A, A>(units) {
     public A d0(A f) { return domain.get(f); }
     public A d1(A f) { return codomain.get(f); }
     public A unit(A x) { return x; }
     public A m(A f, A g) { return m.get(Pair(f, g)); }
     public Set<A> arrows() { return domain.keySet(); }
   };
 }
```

For this method, objects and unit arrows are the same thing. We create objects based on the list of unit arrows, and fill in the tables for domain, codomain and multiplication. Now we have one mor issue: fill in the composition table. If you look at the example in the beginning of this section, it should be obvious what is the composition of **a** and 2.1: it is 1**1**, since it is the only arrow from **1** to **1**. That's what `fillCompositionTable(domain, codomain, m)` does:

```
private static <O, A> void
```

```
   fillCompositionTable(Map<A, O> d0, Map<A, O> d1, Map<Pair<A, A>, A> m) {
  Set<A> arrows = d0.keySet();
  // First, fill in composition table when choice is unique
  for (A f : arrows) for (A g : arrows) {
   if (d1.get(f).equals(d0.get(g))) {
    O d0f = d0.get(f);
    O d1g = d1.get(g);
    A candidate = null;
    boolean unique = true;
    for (A arrow : d0.keySet()) if (d0.get(arrow) == d0f && d1.get(arrow) == d1g) {
     unique = candidate == null;
     candidate = arrow;
    }

    if (unique) {
     m.put(Pair(f, g), candidate);
    }
   }
  }

  for (A f : arrows) for (A g : arrows) for (A h : arrows) {
   if (d1.get(f) == d0.get(g) && d1.get(g) == d0.get(h)) {
    // Here we have three consecutive arrows;
    // we can compose them as f(gh) or as (fg)h;
    // and in case one of these compositions is not defined,
    // we define it right here, since we have enough information
    A gf = m.get(Pair(f, g));
    A hg = m.get(Pair(g, h));
    if (gf != null && hg != null) {
     A h_gf = m.get(Pair(gf, h));
     A hg_f = m.get(Pair(f, hg));
     if (hg_f == null && h_gf != null) m.put(Pair(f, hg), h_gf);
     if (h_gf == null && hg_f != null) m.put(Pair(gf, h), hg_f);
    }
   }
  }
 }
```

With these two methods, the example category above can be built using this:

```
  Category<String,String> three = buildCategory(Set("0", "1", "2"),
    Map(array("0.1", "0.2", "a",  "b", "2.1", "2.a", "2.b",   "2.swap"), // domain
        array("0",   "0",   "1",  "1", "2",   "2",   "2",   "2")),
    Map(array("0.1", "0.2", "a",  "b", "2.1", "2.a", "2.b",   "2.swap"), // codomain
        array("1",   "2",   "2",  "2", "1", "2",   "2",   "2")),
    Map(array(Pair("0.1", "a"), Pair("0.1", "b"), Pair("2.1", "a"), Pair("2.1", "b"), Pair("a",
"2.swap"), Pair("b", "2.swap"), Pair("2.swap", "2.swap")), // composition map
        array(     "0.2",            "0.2",            "2.a",            "2.b",
"b",                 "a",                "2"))
    );
```

# 6. Special Categories

## 6.1. Category "0"

This category does not contain any objects or arrows. Empty. No problems with category laws. Every object in an empty category has an arrow.

In Java it can be expressed like this (probably we need a constant for this):

```
Category<String,String> empty = buildCategory(Set(),
  Map(array(), array()), // domain
  Map(array(), array()), // codomain
     Map(array(), array())  // composition
  )
```

## 6.2. Category "1"

One object, one (unit) arrow. `1 o 1 = 1`.

In Java it can be expressed like this (probably we need a constant for this too):

```
Category<String, String> one = buildCategory(Set("0"),
  Map(array(), array()), // domain
  Map(array(), array()), // codomain
     Map(array(), array())  // composition
  )
```

## Category "2"

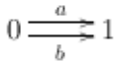
Two objects (0 and 1), three arrows: `10, 11, a : 1 -> 2`. The only nontrivial arrow, a.

In Java it can be expressed like this (probably we need a constant for this too):

```
Category<String, String> two = buildCategory(Set("0", "1"),
  Map(array("a"), array("0")), // domain
  Map(array("a"), array("1")), // codomain
     Map(array(), array())  // composition
  );
```

## Category "Parallel Pair"

Two objects (0 and 1); four arrows: `10, 11, a : 1 -> 2, b: 1 -> 2`. Here's the diagram:



In Java it can be expressed like this (probably we need a constant for this too):

```
 Category<String, String> parallelPair =
  Category(Graph(Set("0", "1"), Map(array("a", "b"), array(Pair("0","1"), Pair("0","1")))),
null);
```

We still do not have to specify any composition.

# III. Examples and Features

## 7. More Categories

In section 6 we saw several examples of categories: **1**, **2**, **Parallel Pair**. Let's discuss more examples here.
Given a segment of integers, [0..n], one can build a category on it. First, the segment can be interpreted as a poset, with the natural order. And a poset can always be interpreted as a category. Note that from categorical point of view, there's no difference between [0..n] and [k..(n+k)], it is just a linearly-ordered set of n elements.
So, in Java, one can introduce the following method:

```
public static final Category<Integer, Pair<Integer, Integer>> segment(int n) {
 return math.cat.Category.Category(PoSet.range(0, n - 1, 1));
}
```

and then redefine segment categories like this:

```
public static final Category<Integer, Pair<Integer, Integer>> _0_ = segment(0);
public static final Category<Integer, Pair<Integer, Integer>> _1_ = segment(1);
public static final Category<Integer, Pair<Integer, Integer>> _2_ = segment(2);
public static final Category<Integer, Pair<Integer, Integer>> _3_ = segment(3);
public static final Category<Integer, Pair<Integer, Integer>> _4_ = segment(4);
```

We can show these categories graphically, like this (omitting unit arrows):

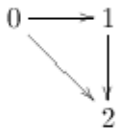
### 7.1. Category _0_

### 7.2. Category _1_

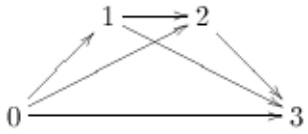
1

### 7.3. Category _2_


### 7.4. Category _3_




### 7.5. Category _4_




### 7.6. Category $Z_2$

Let's take a look at the following category:

```
Z₂ = Category("([0], {1: 0 -> 0, -1: 0 -> 0}), {1 o 1 = 1, 1 o -1 = -1, -1 o 1 = -1, -1 o -1 = 1})");
```

This category has one object and two arrows, 1 (which is unit) and -1, with composition defined so that -1 o -1 = 1. What's remarkable about this category is that it is actually isomorphic to a group of integers modulo 2.

## 8. Bigger Categories

So far we dealt with very small categories: those that could be represented as an object in a program. But look around, there's more.

But wait. First, we have to change the terminology. While the term "arrow" is good when describing our tiny categories, in the world of big categories arrows are called *morphisms.* It is the same thing, but the term *morphism* is traditionally used when we describe arrows between mathematical objects. So we have unit morphisms, morphisms composition, morphisms domains and codomains. Don't panic, it's exactly the same.

### 8.1. Sets

Take any number of sets (from zero to all sets like in Set Theory), and all the functions between them. What do we have: sets as objects, functions as morphisms. Each set has a unit morphism, which is an identity function on itself. Is functions composition associative? By definition.

Note that traditional Java `Map<X, Y>` is not a good candidate for a set function: it is defined on types, `X -> Y`, not on Java `Set`s.

#### *Java Implementation*

Implementing morphisms in Java with generics is more challenging than one would think. You have to parameterize domain and codomain element type, and if you want to subclass, you also have to pass, as generic parameter, the types of domain and codomain objects; and you may also need to pass as a generic parameter the type of the morphism. What's interesting, there would be no such problem if we had closures in Java, which is more of a political, not a theoretical problem these days.

Firstly, let's define a general, category, independent, notion of morphism.

```
public abstract class Morphism<X, Y> {
 private String name;
 private X domain;
 private Y codomain;

 /**
  * Base constructor. Takes domain and codomain
  *
  * @param domain domain of this morphism
  * @param codomain codomain of this morphism
  */
 Morphism(X domain, Y codomain) {
  this.domain = domain;
  this.codomain = codomain;
 }

 /**
  * Constructor.
  *
  * @param name name of this morphism
  * @param domain domain of this morphism
  * @param codomain codomain of this morphism
  */
 Morphism(String name, X domain, Y codomain) {
  this(domain, codomain);
  this.name = name;
 }

 /**
  * @return name of this morphism
  */
 public String name() { return name; }

 /**
  * @return domain of this morphism
  */
 public X domain() { return domain; }

 /**
  * @return codomain of this morphism
  */
 public Y codomain() { return codomain; }

 /**
  * @return String representation of this morphism
  */
 @Override public String toString() { return name == null ? super.toString() : name; }
}
```

One can be tempted to define composition as a method of `Morphism<X,Y>`. There is a tricky problem here. Say you have a morphism `f: A->B`, and you want to compose it with a morphism `g: B -> C`. And you want it to be a method of `f`. This means that `f` morphism should be able to create a new instance of morphism, parameterized by type `C`. Instantiation happens during execution time, when `f` knows nothing about type `C`; and it has no way to create an object that depends on it. That's why we have to (in)conspicuously omit composition here.

Now let's define set morphisms "the right way", that is, the way they should be defined if we treat sets as objects of a category. Buckle up; you'll see a lot of generic parameters here.

```
public abstract class SetMorphism
   <X, // domain data type
    PX extends Set<X>, // domain type
    Y, // codomain data type
    PY extends Set<Y> // codomain type
   > extends Morphism<PX, PY> {

  /**
   * Constructor.
   *
   * @param domain domain of this morphism
   * @param codomain codomain of this morphism
   */
  public SetMorphism(PX domain, PY codomain) {
   super(domain, codomain);
   validate();
  }

  /**
   * Constructor.
   *
   * @param name name of this morphism
   * @param domain domain of this morphism
   * @param codomain codomain of this morphism
   */
  public SetMorphism(String name, PX domain, PY codomain) {
   super(name, domain, codomain);
   validate();
  }

  /**
   * Validates set morphism.
   * All we need is that, each domain element is mapped to a codomain element.
   */
  private void validate() {
   for (X x : domain()) assert(codomain().contains(apply(x))): "Morphism value for "+ x +" out of
d1";
  }

  public String toString() {
   if (name() != null) return name();
   StringBuffer sb = new StringBuffer();
   for (X x : domain()) {
    if (sb.length() > 0) sb.append(", ");
    sb.append(x).append(" -> ").append(apply(x));
   }

   return "{" + sb + "}";
  }

  /**
   * Applies the set morphism, that is a function, to an element of the set
   * @param x element to apply the morphism to
   * @return morphism value on x
   */
  public abstract Y apply(X x);

  /**
   * Two set morphisms are equal if they have equal domains and codomains and map domain elements
```

```
to the same values.
   * Note that there's no negation in this calculation; there is a deep reason for it, to be
disclosed much, much later.
   *
  * @param other set morphism to compare
  * @return true if they are equal
  */
 public boolean equals(SetMorphism<X, PX, Y, PY> other) {
  boolean isEqual = domain().equals(other.domain()) &&
            codomain().equals(other.codomain());
  for (X x : domain()) {
   isEqual = isEqual && apply(x).equals(other.apply(x));
  }
  return isEqual;
 }

 public boolean equals(Object o) { return o instanceof SetMorphism &&
(((SetMorphism)o).equals(this)); }

 /**
  * Factory method. Builds a set morphism, given two sets and a map.
  *
  * @param domain domain set
  * @param codomain codomain set
  * @param map maps domain elements to codomain elements
  * @return a new instance of SetMorphism
  */
 public static <X, Y>
 SetMorphism<X, Set<X>, Y, Set<Y>> Morphism(final Set<X> domain, final Set<Y> codomain, final
Map<X, Y> map) {
  for (X x : domain) {
   assert map.containsKey(x) : "Map should be defined on element " + x;
  }

  return new SetMorphism<X, Set<X>, Y, Set<Y>>(domain, codomain) {
   public Y apply(X x) {
    assert domain.contains(x) : "Argument not in d0";
    return map.get(x);
   }
  };
 }

 /**
  * Factory method. Builds unit morphism for a set (identity function).
  *
  * @param s the set
  * @return identity morphism on the given set
  */
 public static <X> SetMorphism<X, Set<X>, X, Set<X>> unit(Set<X> s) {
  return new SetMorphism<X, Set<X>, X, Set<X>>(s, s) {
   public X apply(X x) {
    return x;
   }
  };
 }

 /**
  * Composes two set morphisms. Note that while SetMorphism can be subclassed to be used by
morphisms of subclasses
  * of Set, composition cannot be defined in a generic way.
  * To make it possible in Java (with generics), one needs closures!
  *
```

```
 * @param f first morphism: X -> Y
 * @param g second morphism: Y -> Z
 * @return their composition g o f: X -> Z
 */
public static <
   X, // domain data type for the first morphism
   Y, // codomain data type for the first morphism (same as domain type for the second morphism)
   Z  // codomain data type for the first morphism
  >
SetMorphism<X, Set<X>, Z, Set<Z>> compose(
   final SetMorphism<X, Set<X>, Y, Set<Y>> f, // X -> Y
   final SetMorphism<Y, Set<Y>, Z, Set<Z>> g  // Y -> Z
 ) {
 assert f.codomain().equals(g.domain()): "Composition not defined";
 return new SetMorphism<X, Set<X>, Z, Set<Z>>(f.domain(), g.codomain()) {
  public Z apply(X x) {
   return g.apply(f.apply(x));
  }
 };
}
}
```

## 8.2. Posets

Similarly, a bunch of posets can be turned into a category if we add, as morphisms, those and only those functions that preserve order.

### *Java Implementation*

Our `PoSet` class `extends AbstractSet`, so let's cannibalize `SetMorphism` to define `PoSetMorphism`. That's not what Effective java recommends.

```
public abstract class PoSetMorphism<X, Y> extends SetMorphism<X, PoSet<X>, Y, PoSet<Y>> {

 public PoSetMorphism(PoSet<X> domain, PoSet<Y> codomain) {
  super(domain, codomain);
  validate();
 }

 public PoSetMorphism(String id, PoSet<X> domain, PoSet<Y> codomain) {
  super(id, domain, codomain);
  validate();
 }

 private void validate() {
  for (X x1 : domain()) for (X x2 : domain()) {
   if (domain()._le_(x1, x2)) {
    assert codomain()._le_(apply(x1), apply(x2));
   }
  }
 }

 public String toString() {
  if (name() != null) return name();
  StringBuffer sb = new StringBuffer();
  for (X x : domain()) {
   if (sb.length() > 0) sb.append(", ");
   sb.append(x).append(" -> ").append(apply(x));
  }
```

```
  return "{" + sb + "}";
 }

 public abstract Y apply(X x);

 public <X, Y> PoSetMorphism<X, Y> PosetMorphism(final PoSet<X> domain, final PoSet<Y> codomain,
final Map<X, Y> map) {
  for (X x : domain) {
   assert map.containsKey(x) : "Map should be defined on d0 element " + x;
  }

  return new PoSetMorphism<X, Y>(domain, codomain) {
   public Y apply(X x) {
    assert domain.contains(x) : "Argument not in d0";
    return map.get(x);
   }
  };
 }

 /**
  * Factory method. Builds unit morphism for a poset (identity function).
  *
  * @param s the set
  * @return identity morphism on the given set
  */
 public static <X> PoSetMorphism<X, X> unit(PoSet<X> s) {
  return new PoSetMorphism<X, X>(s, s) {
   public X apply(X x) {
    return x;
   }
  };
 }

 public static <
   X, // domain data type for the first morphism
   Y, // codomain data type for the first morphism (same as domain type for the second morphism)
   Z  // codomain data type for the first morphism
  >
 PoSetMorphism<X, Z> compose(
   final PoSetMorphism<X, Y> f,
   final PoSetMorphism<Y, Z> g
  ) {
  assert f.codomain().equals(g.domain()): "Composition not defined";
  return new PoSetMorphism<X, Z>(f.domain(), g.codomain()) {
   public Z apply(X x) {
    return g.apply(f.apply(x));
   }
  };
 }
}
```

## 8.3. Graphs

To define a graph morphism, we requure two functions, one for arrows, another for nodes. And these two functions should be consistent: if we have an arrow f: A -> B,
and apply a graph morphism `F = (F0, F1)` (`F0` for nodes, `F1` for arrows), then `F1(f): F0(A) -> F0(B)`, that is, `d0(F1(f)) = F0(d0(f))` and `d1(F1(f)) = F1(d1(f))`.

### *Java Implementation*

Although `Graph` class `extends AbstractSet`, this brings a mixture of convenience and confusion; we better not subclass `GraphMorphism` from `SetMorphism`.

```
public class GraphMorphism<XNodes, XArrows, YNodes, YArrows> extends Morphism<Graph<XNodes,
XArrows>, Graph<YNodes, YArrows>> {
 SetMorphism<XNodes, Set<XNodes>, YNodes, Set<YNodes>> nodesMorphism;
 SetMorphism<XArrows, Set<XArrows>, YArrows, Set<YArrows>> arrowsMorphism;

 /**
  * Constructor. Builds unnamed graph morphism.
  *
  * @param domain domain graph
  * @param codomain codomain graph
  * @param nodesMorphism maps nodes from the first graph to the nodes of the second graph
  * @param arrowsMorphism maps arrows from the first graph to the arrows of the second graph
  */
 public GraphMorphism(
    Graph<XNodes, XArrows> domain,
    Graph<YNodes, YArrows> codomain,
    SetMorphism<XNodes, Set<XNodes>, YNodes, Set<YNodes>> nodesMorphism,
    SetMorphism<XArrows, Set<XArrows>, YArrows, Set<YArrows>> arrowsMorphism) {
  super(domain, codomain);
  this.nodesMorphism = nodesMorphism;
  this.arrowsMorphism = arrowsMorphism;
  validate();
 }

 /**
  * Constructor. Builds a named graph morphism.
  *
  * @param name name of this morphism
  * @param domain domain graph
  * @param codomain codomain graph
  * @param nodesMorphism maps nodes from the first graph to the nodes of the second graph
  * @param arrowsMorphism maps arrows from the first graph to the arrows of the second graph
  */
 public GraphMorphism(
    String name,
    Graph<XNodes, XArrows> domain,
    Graph<YNodes, YArrows> codomain,
    SetMorphism<XNodes, Set<XNodes>, YNodes, Set<YNodes>> nodesMorphism,
    SetMorphism<XArrows, Set<XArrows>, YArrows, Set<YArrows>> arrowsMorphism) {
  super(name, domain, codomain);
  this.nodesMorphism = nodesMorphism;
  this.arrowsMorphism = arrowsMorphism;
  validate();
 }

 /**
  * Validates this graph morphism.
  * A graph morphism is valid if its nodes and arrows components are valid
  * and arrows are mapped consistently: (f:A->B) -> (F(f):F(A)->F(B)).
  */
 private void validate() {
  for (XArrows arrowX : domain().arrows()) {
   XNodes domX = domain().d0(arrowX);
   XNodes codomX = domain().d1(arrowX);
   YArrows arrowY = arrowsMorphism.apply(arrowX);
```

```
    assert nodesMorphism.apply(domX).equals(codomain().d0(arrowY));
    assert nodesMorphism.apply(codomX).equals(codomain().d1(arrowY));
   }
  }

  public int hashCode() {
   return name() != null ? name().hashCode() :
      (domain().hashCode() * 4/*random number*/ + codomain().hashCode());
  }

  /**
   * Two graph morphisms are equal if their domains and codomains are equal, and their acttions on
nodes are the same
   * and their actions on arrows are the same.
   *
   * @param other set morphism to compare
   * @return true if they are equal
   */
  public boolean equals(GraphMorphism<XNodes, XArrows, YNodes, YArrows> other) {
   return domain().equals(other.domain()) &&
       codomain().equals(other.codomain()) &&
       nodesMorphism.equals(other.nodesMorphism) &&
       arrowsMorphism.equals(other.arrowsMorphism);
  }

  public String toString() {
   return name() != null ? name() :
      "(" + nodesMorphism + ", " + arrowsMorphism + ")";
  }

  public static <
     // generic parameters
     XNodes, XArrows, // first graph
     YNodes, YArrows  // second graph
     >
  GraphMorphism<XNodes, XArrows, YNodes, YArrows> GraphMorphism(
     final Graph<XNodes, XArrows> domain,
     final Graph<YNodes, YArrows> codomain,
     final Map<XNodes, YNodes> nodesMap,
     final Map<XArrows, YArrows> arrowsMap) {

   return new GraphMorphism<XNodes, XArrows, YNodes, YArrows>(domain, codomain,
      Morphism(domain.nodes(), codomain.nodes(), nodesMap),
      Morphism(domain.arrows(), codomain.arrows(), arrowsMap));
  }
}
```

## 8.4. Category of Categories

We will take a closer look at it a little bit later.

## 9. Exercises

1. Note that each unit has such a property that when composed (left or right) with another arrow, it leaves that arrow intact. The property defines a unit as a neutral element relative to composition. Prove that for each object there is just one such neutral element.
2. A semigroup is defined as a set of elements with associative multiplication defined on it. A monoid is defined as a semigroup that has a neutral element. Prove that any category with

one object is a monoid.

3. Implement the morphisms of this part in JavaScript.

# IV. Functors and Morphisms

## 10. Functor

In earlier sections we discussed morphisms of big categories, but stopped right before introducing morphisms for categories.
Such morphisms are called *functors.*

### 10.1. Definition

A *functor* `F: A -> B` is an morphism from one category to another; it maps objects of category `A` to objects of category `B`, and arrows of category `A` to arrows of category `B`, so that the following conditions are satisfied:

$F(1_x) = 1_{F(x)}$ for all objects `x` in `A`;
`F(d0(f)) = d0(F(f))` for all arrows `f: x -> y` in `A`;
`F(d1(f)) = d1(F(f))` for all arrows `f: x -> y` in `A`;
`F(g o f) = F(g) o F(f)` for all arrows `f: x -> y` and `g : y -> z` in `A`.

These conditions just assert that units, multiplication, domain and codomain are preserved by the functors.

### 10.2. Examples

The simplest example of a functor is an identity functor, $1_C$ `: C -> C` that leaves everything intact in category `C`.

Another simple example is this: take a category **`_1_`**; it has just one object and one arrow. So to define a functor `F: _1_ -> C`, one has to map the object and the unit arrow. Unit arrow should map to the unit, so all we need is to specify an object. On the other hand, take any object `x` in category `C`; it obviously defines a functor from `_1_` to `C`.

For each category `C` there is exactly one functor from `C` to **`_1_`**: it maps every object of `C` to `0` and every arrow of `C` to `10`.

A little bit more complicated example of a functor is when we map `_2_` to a category `C`. _2_ has two objects and three arrows. Two of these arrows are units, so they will map to units. And the remaining arrow maps to another arrow that connects images of the two objects:

(`0->1`) ---> (`f: x->y`). There are no additional requirements; and, as you can see, any arrow `f: x->y` in a category `C` defines a functor from _2_ to `C`.

You can try to play with other sample categories and see how you can define a functor on such a category. Later we will return to this topic.

### 10.3. Java Implementation

This code looks a little bit scary; I never saw so many generic parameters in my life; but essentially it is very straightforward. What we need is a `GraphMorphism` that preserves units and composition:

```
/**
```

```
 * Functor class: morphisms for categories.
 * All code is <a href="http://myjavatools.com/projects/Category/source.zip">here</a>
 */
public class Functor<
   XObjects, // type of nodes in the first category (like Alksnis?)
   XArrows, // type of arrows in the first category
   YObjects, // type of nodes in the second category
   YArrows   // type of arrows in the second category
   > extends GraphMorphism<XObjects, XArrows, Category<XObjects, XArrows>, YObjects, YArrows,
Category<YObjects, YArrows>> {

  /**
   * Constructor. Builds unnamed functor.
   *
   * @param domain domain category
   * @param codomain codomain category
   * @param objectsMorphism maps objects from the first category to the objects of the second
category
   * @param arrowsMorphism maps arrows from the first category to the arrows of the second
category
   */
  public Functor(
     Category<XObjects, XArrows> domain,
     Category<YObjects, YArrows> codomain,
     SetMorphism<XObjects, Set<XObjects>, YObjects, Set<YObjects>> objectsMorphism,
     SetMorphism<XArrows, Set<XArrows>, YArrows, Set<YArrows>> arrowsMorphism) {
   super(domain, codomain, objectsMorphism, arrowsMorphism);
   validate();
  }

  /**
   * Constructor. Builds named functor.
   *
   * @param name name of this functor
   * @param domain domain category
   * @param codomain codomain category
   * @param objectsMorphism maps objects from the first category to the objects of the second
category
   * @param arrowsMorphism maps arrows from the first category to the arrows of the second
category
   */
  public Functor(
     String name,
     Category<XObjects, XArrows> domain,
     Category<YObjects, YArrows> codomain,
     SetMorphism<XObjects, Set<XObjects>, YObjects, Set<YObjects>> objectsMorphism,
     SetMorphism<XArrows, Set<XArrows>, YArrows, Set<YArrows>> arrowsMorphism) {
   super(name, domain, codomain, objectsMorphism, arrowsMorphism);
   validate();
  }

  /**
   * Validates this functor.
   * A functor is valid if it is valid as a graph morphism, and besides,
   * it preserves unit and arrows composition.
   * That is, F(unit(x)) == unit(F(x)), and
   * F(g) o F(f) = F(g o f)
   */
  private void validate() {
   for (XObjects x : domain().objects()) {
    XArrows ux = domain().unit(x);
```

```
    YObjects y = nodesMorphism.apply(x);
    YArrows uy = codomain().unit(y);
    assert uy.equals(arrowsMorphism.apply(ux)) :
      "Functor must preserve units (failed on " + x + ")";
   }

   for (XArrows fx : domain().arrows()) for (XArrows gx : domain().arrows()) {
    if (domain().d1(fx).equals(domain().d0(gx))) {
     XArrows fx_gx = domain().m(fx, gx);
     YArrows fy = arrowsMorphism.apply(fx);
     YArrows gy = arrowsMorphism.apply(gx);
     YArrows fy_gy = codomain().m(fy, gy);
     assert fy_gy.equals(arrowsMorphism.apply(fx_gx)) :
       "Functor must preserve composition (failed on " + fx + ", " + fy + ")";
    }
   }
  }


  /**
   * Factory method. Builds unit functor for a category (identity functor).
   *
   * @param c the category
   * @return identity functor on the given category
   */
  public static <XObjects, XArrows>
   Functor<XObjects, XArrows, XObjects, XArrows> unit(Category<XObjects, XArrows> c) {
   return new Functor<XObjects, XArrows, XObjects, XArrows>
     (c, c, SetMorphism.unit(c.objects()), SetMorphism.unit(c.arrows()));
  }

  /**
   * Composes two functors
   * @param f : X -> Y - first functor
   * @param g : Y -> Z - second functor
   * @return g o f : X -> Z - their composition
   */
  public static <
    XObjects,  // nodes type for the category in the chain
    XArrows, // arrows type for the first category in the chain
    YObjects,  // nodes type for the second category in the chain
    YArrows, // arrows type for the second category in the chain
    ZObjects,  // nodes type for the third category in the chain
    ZArrows  // arrows type for the third category in the chain
   >
  Functor<XObjects, XArrows, ZObjects, ZArrows>
  compose(
    final Functor<XObjects, XArrows, YObjects, YArrows> f,
    final Functor<YObjects, YArrows, ZObjects, ZArrows> g
   ) {
   assert f.codomain().equals(g.domain()): "Composition not defined";
   return new Functor<XObjects, XArrows, ZObjects, ZArrows>(
     f.domain(),
     g.codomain(),
     SetMorphism.compose(f.nodesMorphism, g.nodesMorphism),
     SetMorphism.compose(f.arrowsMorphism, g.arrowsMorphism));
  }

  /**
   * Builds a functor, given two categories and two maps, one for the set of nodes, the other for
the set of arrows.
```

```
  *
  * @param domain first category
  * @param codomain second category
  * @param nodesMap maps nodes of the first category to nodes of the second category
  * @param arrowsMap maps arrows of the first category to arrows of the second category
  * @return a functor that encapsluates all this
  */
 public static <
    // generic parameters
    XNodes, XArrows, // first category
    YNodes, YArrows  // second category
    >
 Functor<XNodes, XArrows, YNodes, YArrows> Functor(
    final Category<XNodes, XArrows> domain,
    final Category<YNodes, YArrows> codomain,
    final Map<XNodes, YNodes> nodesMap,
    final Map<XArrows, YArrows> arrowsMap) {

  return new Functor<XNodes, XArrows, YNodes, YArrows>(
     domain,
     codomain,
     Morphism(domain.nodes(), codomain.nodes(), nodesMap),
     Morphism(domain.arrows(), codomain.arrows(), arrowsMap));
 }
}
```

The code in svn (let me know if you can't access it) has a unittest for this class.

# 11. Kinds of Arrows

Now we return to our small categories and take a closer look at the arrows (aka morphisms). We already know that there is a very special kind of arrows: unit arrows. They end where they start, and they are neutral for composition.

## 11.1. Endomorphism

An arrow that has the same domain and codomain is called endomorphism. If you look at the category $\mathbf{Z_2}$, both its arrows are endomorphisms. If you take a set of, say, 3 elements, {0, 1, 2}, in a category `Set`, how many endomorphisms can we define on such set? `33`, of course; some of them (namely, permutations) will be *isomorphisms* (see below), some not.

## 11.2. Isomorphism

Let `f : x -> y` and `g : y -> x` be two arrows. If $g \circ f = 1_x$ and $f \circ g = 1_y$, then `g` is an *inverse* arrow for `f`, and `f` is an *inverse* arrow for `g`; this is also denoted as $g = f^{-1}$ and $f = g^{-1}$. An arrow that has an inverse is called isomorphism. So, we have two isomorphisms, `f` and `g`; and the two objects, `x` and `y`, are called *isomorphic*. Namely, two objects are *isomorphic* if there is an *isomorphism* between the two.

It is clear that the relationship of being isomorphic is reflexive (`x` is isomorphic to `x`), symmetric (if `x` is isomorphic to `y`, then `y` is isomorphic to `x`), and transitive (`x` isomorphic to `y`, `y` isomorphic to `z` yields `x` is isomorphic to `z`).

In category $\mathbf{Z_2}$ the arrow "-1" is an isomorphism: its inverse is itself.

Note that if two objects are isomorphic it does not mean that there is some kind of canonical

isomorphism between the two. On some occasions this is the case, on others - no. There can be a dozen of isomorphisms between two objects. Like, say, between segments of integers [0..2] and [1..3] in `Set`: both contain four elements, so they are isomorphic, but there are 6 different isomorphisms each way.

## 11.3. Java Implementation

Let's add this to `Category` class:

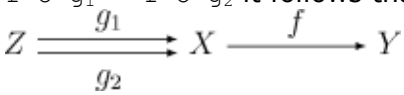
```
private Map<A, A> inverse = new HashMap<A, A>(); // cache expensive results

 public A inverse(A arrow) {
   if (inverse.containsKey(arrow)) return inverse.get(arrow);

   for (A candidate : arrows(d1(arrow), d0(arrow))) {
     if (m(arrow, candidate).equals(unit(d0(arrow))) &&
         m(candidate, arrow).equals(unit(d1(arrow)))) {
       inverse.put(arrow, candidate);
       return candidate;
     }
   }
   inverse.put(arrow, null);
   return null;
 }

 public boolean isIsomorphism(A arrow) {
   return inverse(arrow) != null; // there should be a better way to detect existence
 }
```

## 11.4. Monomorphism

Isomorphism was easy to define in terms of arrows; but how about what is known as "injective function" - how can we define it in a category without referring to "elements"? The trick is to use other arrows instead of elements. An arrow `f: X -> Y` acts on arrows ending at `X` by turning them into arrows ending at `Y`: each `g: Z -> X`
maps to an arrow `f o g: Z -> Y`; so we can define a monomorphism based on injectivity of this map.

### *Definition*

An arrow `f: X -> Y` is called a monomorphism if for each parallel pair $g_1$`: Z -> X,` $g_2$`: Z -> X` from `f o` $g_1$ `= f o` $g_2$ it follows that $g_1 = g_2$.




### *Java Implementation*

```
private Map<A, Boolean> monomorphisms = new HashMap<A, Boolean>();

public boolean isMonomorphism(A arrow) {
 if (monomorphisms.containsKey(arrow)) return monomorphisms.get(arrow);

 O x = d0(arrow);
 boolean result = true;
 for (A f1 : arrows()) if (result && d1(f1).equals(x)) {
  for (A f2 : arrows(d0(f1), x)) {
   if (m(f1, arrow).equals(m(f2, arrow))) result = result && f1.equals(f2);
  }
 }
 monomorphisms.put(arrow, result);
```

```
  return result;
 }
```

## 11.5. Epimorphism

An epimorphism is a notion dual to monomorphism.

### *Definition*

An arrow `f: X -> Y` is called a epimorphism if for each parallel pair $g_1$`: Y -> Z,` $g_2$`: Y -> Z` from $g_1 \circ f = g_2 \circ f$ it follows that $g_1 = g_2$.

### *Java Implementation*

```
private Map<A, Boolean> epimorphisms = new HashMap<A, Boolean>();

public boolean isEpimorphism(A arrow) {
 if (epimorphisms.containsKey(arrow)) return epimorphisms.get(arrow);

 O y = d1(arrow);
 boolean result = true;
 for (A f1 : arrows()) if (result && d0(f1).equals(y)) {
  for (A f2 : arrows(y, d1(f1))) {
   if (m(arrow, f1).equals(m(arrow, f2))) result = result && f1.equals(f2);
  }
 }
 epimorphisms.put(arrow, result);
 return result;
}
```

# 12. $C^{op}$

Say we have a category `C`. We can build an *opposite* category to `C` by reverting the direction of arrows. This new category is called `Cop`.
It is more or less clear that monomorphisms become epimorphisms, and epimorphisms turn into monomorphisms via such an operation. Isomorphisms are defined in a symmetric way, so they remain isomorphisms.

`Cop` helps to explain the notion of "variance". What was introduced in section 10 as a functor is also known as a "covariant functor". There is also a contravariant functor. Say, we have a (covariant) functor `F: A -> Bop`; what does it mean? It means that F maps objects of `A` to objects of `B`, and morphisms of `A` to morphisms of `B`, preserving units, but reverting (relative to category `B`) the direction of arrows: for `f: x -> y` in `A` we have `F(f): F(y) -> F(x)` in `B`.

As an example, we could take the functor that maps a set `x` to its powerset, `2x`. a function `f: x -> y` is mapped to `2f: 2x -> 2y` which, for each $a \in x$, produces $f^{-1}(a) \in y$. As an exercise, prove that this is a correctly defined functor.

## 12.1. Note For Programmers

Yes, this is the same co/contravariance that we encounter in Java generics; say, `Map<X, Y>` is covariant by Y and contravariant by `X`; but, contrary to an easy explanation, it is not the direct consequence of "map being something like $Y^X$", but is determined by the signatures of the methods of this interface.

## 13. Cat

This is the category of categories. Unlike the infamous set of sets, it has all the rights to exist. There is no foundation axiom or separation axiom, so it's perfectly okay. Attempts to build a formal "set-like" category of categories can be found here.

In **Cat**, objects are categories and arrows are functors between these categories. Small creatures like **_1_**, **_2_**, **_3_** are there, together with such monsters as Set, $\mathtt{Set}^{op}$, $\mathtt{Set}_f$ (all finite sets), as well as Top (all topological spaces), Grp (all groups). An archetypical example of a functor is "discrete something" from, say, Set to Top or Cat; it maps a set to a discrete topological space or category (a discrete category is a category whose only arrows are units). Another example is a forgetful functor that maps, say, categories to sets of their objects, or topological spaces to sets of their points. These are important kinds of functors, and I hope to return to them later.

# V. Some Limits

## 14. Java Again

There is an opinion that Java may be not the best language to use while implementing categories. But Java has some advantages, too. First, I know it; second, it is an extremely stable language. And anyway, years ago I was already implementing categories in Fortran; Java is better.

In short, the tricks are these:

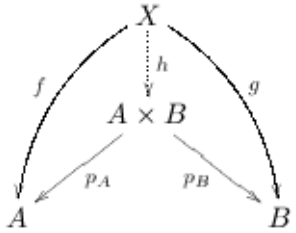
- set product is added to Base class;
- a new class, Predicate, was introduced;
- this new `Predicate` class, unlike its popular analogs, is rich in functionality: it has the following methods that take a set:
  - `find : Set<X> -> Set<X>`, builds a virtual set that contains only elements satisfying the predicate;
  - `forall : Set<X> -> boolean`, checks if the predicate returns true for all elements of a set;
  - `findOne : Set<X> -> X`, returns an arbitrary element of a given set that satisfies the predicate (`null` if none found);
  - `exists : Set<X> -> boolean`, checks if there is an element in the set that satisfies the predicate;
  - `existsUnique : Set<X> -> boolean`, checks if there is exactly one element in the set that satisfies the predicate;
- in categories code, I tend to define predicate factories instead of just methods returning boolean values: one can do nothing with boolean-valued methods, but with predicates one can easily combine them to build complex formulas. See below for specific examples.

## 15. Cartesian Product

You are familiar with a notion of Cartesian product of sets: take two sets, `A` and `B`; their product, `A×B`, is a set of pairs, `{(a, b) | a∈A, b∈B}`. We have to generalize this notion to generic categories, where objects are not necessarily defined by their elements.

### 15.1. Definition

Given two objects, `A` and `B`, their *Cartesian product* is an object `A×B` with two *projection* morphisms $p_A$`: A×B -> A` and $p_B$`: A×B -> B`, such that for each pair `f: X -> A` and `g: X -> B` there is a *unique* morphism `h: X -> A×B` such that `f =` $p_A$ `∘ h` and `g =` $p_B$ `∘ h`.



## 15.2. Properties

**A Cartesian product of two objects is unique up to an isomorphism.** How is it and what does it mean? Let's say we have two objects, `C` and `D`, that satisfy the definition of Cartesian product for objects `A` and `B`. Let's give names to their projections to `A` and `B`, $p_{CA}$: `C -> A`, $p_{CB}$: `C -> B`, $p_{DA}$: `D -> A`, $p_{DB}$: `D -> B`.
Then there are two unique morphisms, `h: C -> D` and `k: D -> C`, that satisfy the conditions from the definition:
$p_{DA} \circ h = p_{CA}$, $p_{DB} \circ h = p_{CB}$, $p_{CA} \circ k = p_{CB}$, $p_{DB} \circ k = p_{CB}$.

Now the composition `l = h o k: D -> D` is a unique morphism that satisfies the conditions $p_{DA} \circ l = p_{DA}$, $p_{DB} \circ l = p_{DB}$. But we already know such a morphism, it is $1_D$. Being unique, it is the same, $l = 1_D$. So we have $h \circ k = 1_D$; similarly $k \circ h = 1_C$. This means that `k` and `h` are isomorphisms; being unique, they are *canonical isomorphisms*.

Note that projections $p_A$: `A×B -> A` and $p_B$: `A×B -> B` are not necessarily epimorphisms. E.g. in Sets, if `A` is an empty set, the product will be empty, but `B` does not have to be empty.

## 15.3. Discussion

Is this definition the same as "the set of all pairs" in Sets?
First, let's check if for two sets, `A` and `B`; a set of pairs, `C = {(a, b) | a∈A, b∈B}` has the universal property 1. If we have `f: X -> A` and `g: X -> B`, then `h: X -> C` is determined by the property, $f = p_A \circ h$ and $g = p_B \circ h$: it is the same as saying that `h(x) = (f(x), g(x)`.
Second, since the product is defined up to an isomorphism, any such product set will be isomorphic to the set of pairs.

## 15.4. Java Implementation

```
public Pair<A, A> product(final O x, final O y) {
 return new Predicate<Pair<A, A>>() {
  public boolean eval(Pair<A, A> p) {
   return isProduct(p, x, y);
  }
 }.findOne(setProduct(arrows(), arrows()));
}
```

This method returns two projection morphisms; the product object itself is their (common) domain. If the product does not exist, `null` is returned. We have to define another method, the one that detects whether an object (with a pair of projections) is a product of two objects:

```
public boolean isProduct(final Pair<A, A> p, final O x, final O y) {
 final O prod = d0(p.x());
 return
   d0(p.y()).equals(prod) &&
   d1(p.x()).equals(x) &&
   d1(p.y()).equals(y) &&
   factorsUniquelyOnTheRight(p).forall(pairsOfArrowsFromSameDomainTo(x, y));
}
```

This method checks the projections' domains and codomains and make sure that any pair of morphisms from the same domain to `x` and `y` factors uniquely throw the given pair of projections. I am not going to write down all the code here, here is the method that returns the set of pairs of arrows from one domain to `x` and `y`:

```
public Set<Pair<A, A>> pairsOfArrowsFromSameDomainTo(final O x, final O y) {
 return new Predicate<Pair<A, A>>() {
  public boolean eval(Pair<A, A> p) {
   return d0(p.x()) == d0(p.y()) && d1(p.x()).equals(x) && d1(p.y()).equals(y);
  }
 }.find(setProduct(arrows(), arrows()));
}
```

*(Note how (relatively) convenient it is to express everything using predicates.)*

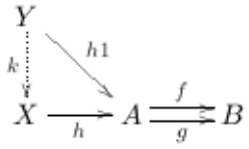
# 16. Equalizer

This is the first example of a limit; the general definition will come later.

## 16.1. Definition

Given a parallel pair of morphisms, `f,g: A -> B`, their equalizer is a morphism `h: X -> A`, such that

1. `h` equalizes `f` and `g`: `f o h = g o h`;
2. `h` is *universal* among morphisms equalizing `f` and `g`: for each $h_1$`: Y -> A` such that `f o` $h_1$ `= g o` $h_1$ there is a *unique* morphism `k: Y -> X` so that $h_1$ `= h o k`.

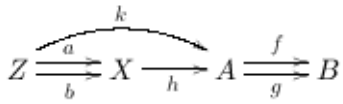



## 16.2. Example

Let `A = B = N`, the set of natural numbers; and let `f(n) = n, g(n) = n * n`. Then the equalizer is a two-element subset of `N`, `{0,1} ∈ N`.

## 16.3. Properties

**An equalizer is always a monomorphism.** See, if we take two morphisms, `a,b: Z -> X` such that `h o a = h o b` (let's denote it as `k`), then `k` equalizes `f` and `g`.
From which it follows that there is a *unique* `c: Z -> X` such that `h o c = k`; since `h o a = k` and `h o b = k`, both `b = c` and `a = c`. The following diagram illustrates it:




**An equalizer of two morphisms is unique up to an isomorphism.** Suppose there are two equalizers, $h_1$ and $h_2$. By definition there must be a unique $k_1$ such that $h_1 = h_2 \circ k_1$, and a unique $k_2$ such that $h_2 = h_1 \circ k_2$; so we have $h_1 = h_1 \circ k_2 \circ k_1$. $h_1$ being a monomorphism, $k_2 \circ k_1 = 1$. The same goes for $k_1 \circ k_2$; so they are inverse to each other.

## 16.4. Java Implementation

What we need is to scan all the morphisms (would be nice if we could delimit our search with only morphisms that end when the two parallel morphisms start), and find the unique arrow that is an

equalizer of the two.

```
public A equalizer(final A f, final A g) {
 assert d0(f) == d0(g) : "Pair should be parallel";
 assert d1(f) == d1(g) : "Pair should be parallel";
 return isEqualizer(f, g).findOne(arrows());
}
```

Here we find an arrow (a morphism) that is an equalizer for `f` and `g`. How do we know it is an equalizer? Any morphism that equalizes `f` and `g` should go through `h`.

```
public Set<A> equalizingArrows(final A f, final A g) {
 return new Predicate<A>() {
  public boolean eval(A a) {
   return d1(a) == d0(f) && m(a, f) == m(a, g);
  }
 }.find(arrows());
}
```

This produces a set of all morphisms that equalize `f` and `g`; now we have to say, in Java, that each morphism of this set factors through `h`:

```
public Predicate<A> isEqualizer(final A f, final A g) {
 return new Predicate<A>() {
  public boolean eval(final A h) {
   return
     d1(h) == d0(f) &&
     d1(h) == d0(g) &&
     m(h, f) == m(h, g) &&
     factorsUniquelyOnLeft(h).forall(equalizingArrows(f, g));
  }
 };
}
```

A morphism `u: x -> z` "factorsUniquelyOnLeft" by `h: y -> z` means literally that there is just one such `v: x -> y` such that `h o v = u`. This predicate is also defined in the code; you can check it out in Category.java in the source code.

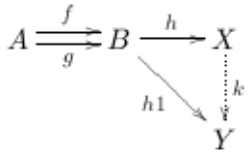
# 17. Coequalizer

Coequalizer is a notion dual to equalizer.

## 17.1. Definition

Given a parallel pair of morphisms, `f,g: A -> B`, their coequalizer is a morphism `h:B -> X`, such that

1. `h` coequalizes `f` and `g`: `h o f = h o g`;
2. `h` is *universal* : for each `h1: B -> Y` such that `h1 o f = h1 o g` there is a *unique* morphism `k: X -> Y` such that `h1 = k o h`.



## 17.2. Example

Let `A = B = Z`, the set of natural numbers; and let `f(n) = n, g(n) = n + 3`. Then the coequalizer is a three-element set that represents three classes of integers modulo 3: {0, 1, 2}. How is it? The trick is that in the coequalizer morphism, Z -> Z3, merges all pairs of numbers (n, n+3); as a result, -3, 0, 3, 6, ... must map to one value (call it 0); -2, 1, 4, 7,... must map to another value (call it 1), and -1, 2, 5, 8, ... must map to yet another value (call it 2).

## 17.3. Properties

**A coequalizer is always an epimorphism.** The proof is dual to the proof of the fact that an equalizer is a monomorphism.

**A coequalizer of two morphisms is unique up to an isomorphism.** The proof is dual to the similar fact for equalizers.

## 17.4. Java Implementation

```
public boolean isCoequalizer(final A h, final A f, final A g) {
 return
   d0(h) == d1(f) &&
   d0(h) == d1(g) &&
   m(f, h) == m(g, h) &&
   factorsUniquelyOnRight(h).forall(coequalizingArrows(f, g));
}
```

where

```
private Set<A> coequalizingArrows(final A f, final A g) {
 return new Predicate<A>() {
  public boolean eval(A a) {
   return d0(a) == d1(f) && m(f, a) == m(g, a);
  }
 }.find(arrows());
}
```

and

```
public boolean isCoequalizer(final A h, final A f, final A g) {
 return
   d0(h) == d1(f) &&
   d0(h) == d1(g) &&
   m(f, h) == m(g, h) &&
   factorsUniquelyOnRight(h).forall(coequalizingArrows(f, g));
}
```

# VI. Pullback

## 18. Pullback

### 18.1. Philosophy

The notion of Cartesian product, introduced in the previous part, is pretty familiar to everyone; not only from set theory, but from sql too. In sql it would look something like this:

```
select * from A, B;
```

Of course you will hardly encounter such a statement in real life; in real life it would look rather like this:

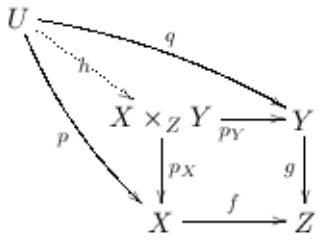
```
select (a,b) from A, B where f(a) = g(b);
```

with `f()` and `g()` defined in some acceptable way. This statement extracts only a part of $A \times B$, those elements that satisfy our condition. And this is the essence of pullback, which is one more example of a limit in category.

### 18.2. Definition

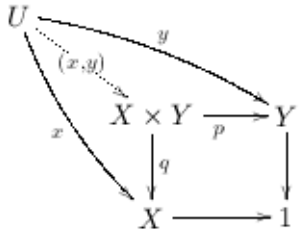
(Before giving the definition, let's talk about notation. Pullback of two objects, $X$ and $Y$, over $Z$, is denoted as $X \times_Z Y$ - it is almost the same as Cartesian product $X \times Y$, but "taken over $Z$". So treat it as a product of $X$ and $Y$ over $Z$.)

Given two morphisms, $f: X \to Z$ and $g: Y \to Z$, their *pullback*, denoted as $X \times_Z Y$, is an object ($X \times_Z Y$) and two morphisms, $p_X: X \times_Z Y \to X$ and $p_Y: X \times_Z Y \to Y$, such that $f \circ p_X = g \circ p_Y$, and they are *universal* regarding this property: for each $p: U \to X$, $q: U \to Y$ such that $f \circ p = g \circ q$, there is a unique $h: U \to X \times_Z Y$ such that $p = p_X \circ h$ and $q = p_Y \circ h$ (see the diagram).

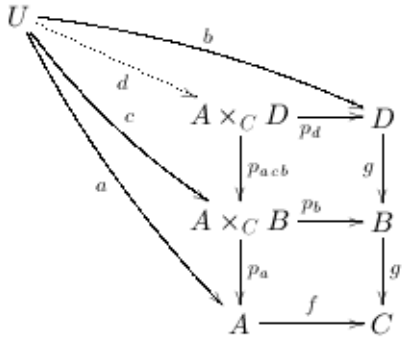



### 18.3. Properties

Some categories have a "terminal object", denoted as $\mathbf{1}$. Each object of a category has just one morphism to $\mathbf{1}$. So, if such an object exists, a pullback of a diagram where $Z$ is 1, is just a Cartesian product:




Suppose we connect two pullback diagrams together, like in the picture:




Is the pullback of $A \times_C B$ and $D$ actually the same as the pullback $A \times_C D$? The answer is yes. The proof is an exercise. You can actually figure it out just looking at the picture above.

## 18.4. Java Code

```
public Pair<A, A> pullback(final A f, final A g) {
 assert d1(f).equals(d1(g)) : "Codomains should be the same";
   return new Predicate<Pair<A, A>>() {
     public boolean eval(Pair<A, A> p) {
       return isPullback(p, f, g);
  }
   }.findOne(setProduct(arrows(), arrows()));
 }
```

And we have to define the predicate that checks whether a pair is a pullback:

```
public boolean isPullback(final Pair<A, A> p, final A f, final A g) {
 final O pullback = d0(p.x());
 return
    d0(p.y()).equals(pullback) &&
    d1(p.x()).equals(d0(f)) &&
    d1(p.y()).equals(d0(g)) &&
    m(p.x(), f).equals(m(p.y(), g)) &&
    factorsUniquelyOnTheRight(p).forall(pairsEqualizing(f, g));
}

public Set<Pair<A, A>> pairsEqualizing(final A f, final A g) {
 return new Predicate<Pair<A, A>>() {
  public boolean eval(Pair<A, A> p) {
   return
      d0(p.x()).equals(d0(p.y())) &&
      canCompose(p.x(), f) &&
      canCompose(p.y(), g) &&
      m(p.x(), f).equals(m(p.y(), g));
  }
 }.find(setProduct(arrows(), arrows()));
}
```

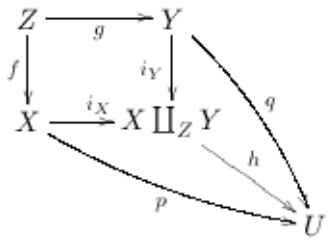
```
private boolean canCompose(A f, A g) {
 return d1(f).equals(d0(g));
}
```

# VII. Pushout and Union

## 19. Pushout

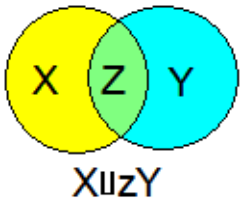
### 19.1. Definition

Given two morphisms, `f: Z -> X` and `g: Z -> Y`, their pushout, denoted as $X \coprod_Z Y$, is an object ($X \coprod_Z Y$) and two morphisms, $i_X$: `X -> X` $\cup_Z$ `Y` and $i_Y$: `Y -> X` $\cup_Z$ `Y`, such that $i_X \circ f = p_Y \circ g$, and they are *universal* regarding this property: for each `p: X -> U`, `q: Y -> U` such that `p o f = q o g`, there is a unique `h: X` $\coprod_Z$ `Y -> U` such that `p = h o` $i_X$ and `q = h o` $i_Y$ (see the diagram).




### 19.2. Philosophy

The notion of pushout is dual to pullback. In case when `Z` is a subobject of `X` and `Y` (that is, when `f` and `g` are monomorphisms), pushout can be easily illustrated as a union of two objects having a common subobject:




### 19.3. Java Code

```
public Pair<A, A> pushout(final A f, final A g) {
   assert d0(f).equals(d0(g)) : "Domains should be the same";
   return new Predicate<Pair<A, A>>() {
     public boolean eval(Pair<A, A> p) {
       boolean result = isPushout(p, f, g);
       trace("isPullback?(", p, ",", f, ",", g, ")->", result);
       return result;
     }
   }.findOne(setProduct(arrows(), arrows()));
```

```
}
```

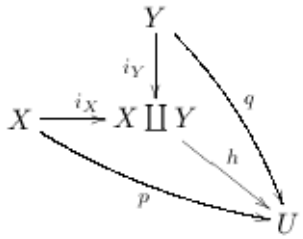
And we have to define the predicate that checks whether a pair is a pushout:

```
private boolean isPushout(Pair<A, A> p, A f, A g) {
   final O pushoutObject = d1(p.x());
   return
       d1(p.y()).equals(pushoutObject) &&
       d0(p.x()).equals(d1(f)) &&
       d0(p.y()).equals(d1(g)) &&
       m(f, p.x()).equals(m(g, p.y())) &&
       factorsUniquelyOnTheRight(p).forall(pairsCoequalizing(f, g));
}
```

# 20. Union

## 20.1. Definition

Given two objects, `X` and `Y`, their *(disjoint) union*, denoted as `X ∐ Y`, is an object (`X ∐ Y`) and two *universal* morphisms, `iX: X -> X ∐ Y` and $i_Y$`: Y -> X ∐ Y`: for each `p: X -> U`, `q: Y -> U` there is a unique `h: X ∐ Y -> U` such that `p = h o` $i_X$ and `q = h o` $i_Y$ (see the diagram).




## 20.2. Java Code

```
public Pair<A, A> union(final O x, final O y) {
  return new Predicate<Pair<A, A>>() {
    public boolean eval(Pair<A, A> p) {
      return isUnion(p, x, y);
    }
  }.findOne(setProduct(arrows(), arrows()));
}
```

And we have to define the predicate that checks whether a pair is a union:

```
public boolean isUnion(final Pair<A, A> i, final O x, final O y) {
   final O unionObject = d1(i.x());
   return
       d1(i.y()).equals(unionObject) &&
       d0(i.x()).equals(x) &&
       d0(i.y()).equals(y) &&
       factorsUniquelyOnTheLeft(i).forall(pairsWithTheSameCodomain(x, y));
}
```

# VIII. 0, 1, $X^n$

## 21. Terminal Object

### 21.1. Definition

**1**, the *terminal object*, is such an object in a category that every object has exactly one arrow to **1**.

### 21.2. Philosophy

If you think in the terms of sets, **1** is the singleton set. Which singleton set, you may ask? Does not matter; as any other limit, **1** is defined up to an isomorphism; it is obvious that all singleton sets are isomorphic.

In `Sets` any object can be constructed as a union (see section 20) of **1**s. This is not so in less trivial categories, small or large; for instance, in **Poset** a union of **1**s is a discrete poset; two posets may contain the same elements but be ordered differently, hence be different, non-isomorphic objects.

### 21.3. Java Code

```
public Pair<A, A> pushout(final A f, final A g) {
   assert d0(f).equals(d0(g)) : "Domains should be the same";
   return new Predicate<Pair<A, A>>() {
     public boolean eval(Pair<A, A> p) {
       boolean result = isPushout(p, f, g);
       trace("isPullback?(", p, ",", f, ",", g, ")->", result);
       return result;
     }
   }.findOne(setProduct(arrows(), arrows()));
}
```

And we have to define the predicate that checks whether a pair is a pushout:

```
private boolean isPushout(Pair<A, A> p, A f, A g) {
   final O pushoutObject = d1(p.x());
   return
       d1(p.y()).equals(pushoutObject) &&
       d0(p.x()).equals(d1(f)) &&
       d0(p.y()).equals(d1(g)) &&
       m(f, p.x()).equals(m(g, p.y())) &&
       factorsUniquelyOnTheRight(p).forall(pairsCoequalizing(f, g));
 }
```

## 22. $X^n$

We probably do not even need a formal definition for $X^n$: we already have all the necessary tools to build it.
Let $X^0$ be $1$, $X^1$ be $X$, and $X^n = X^{n-1} \times X$. If we were dealing with sets, $X^n$ would be a set of n-tuples of elements of $X$; it is not as trivial in a generic case.

### 22.1. Java Code

```
public Pair<O, List<A>> degree(O x, int n) {
  assert n >= 0 : "Degree should be positive, we have " + n;
  if (n == 0) return Pair(terminal(), (List<A>) Collections.EMPTY_LIST);
  if (n == 1) return Pair(x, Arrays.asList(this.unit(x)));
  Pair<O, List<A>> previous = degree(x, n - 1);
  Pair<A, A> pq = product(x, previous.x());
  List<A> projections = new ArrayList<A>();
  projections.add(xn.x());
  for (A a : previous.y()) {
    projections.add(m(xn.y(), a));
  }
  return Pair(d0(xn.x()), projections);
}
```

Let's take a look what's going on here. First, what does this method return? It returns a pair, the first element of which is a degree object, $X^n$, and the second is the list of projections of this degree object to the individual instances of $X$ (there must be $n$ such projections).

Now how do we do it? Forget about the three trivial cases. What if we have the degree $n > 1$? First, we produce $X^{n-1}$. It is represented as a pair: object and the list of projections.

How do we build an $X^n$ out of it? Multiply it with X (order does not matter... up to an isomorphism). So we build a Cartesian product, and obtain a pair of projections $p: X^n \to X$ and $q : X^n \to X^{n-1}$. (In the code this pair is denoted as pq.) p's domain is the object $X^n$, and this is the first element of the pair we have to return.

The second element is more challenging. We need to build a list of projections from $X^n$ to $X$. Again, its first element is easy, it is $p$. The next $(n-1)$ elements are built as a composition $p_i: X^n \to X^{n-1} \to X$.

# 23. Initial Object

## 23.1. Definition

**0**, the *initial object,* is such an object in a category that has exactly one arrow to every object.

## 23.2. Philosophy

In Sets, an empty set is an initial object. Unlike **1**, it is unique.

## 23.3. Java Code

```
final public Predicate<O> isInitial = new Predicate<O>() {
  public boolean eval(final O candidate) {
    return new Predicate<O>() {
      public boolean eval(O x) {
        return arrows(candidate, x).size() == 1;
      }
    }.forall(nodes());
  }
};

private O initial = null;
private boolean initialFound = false;
```

```
public O initial() {
  if (!initialFound) {
    initial = isInitial.findOne(nodes());
    initialFound = true;
  }
  return initial;
}
```

# IX. Diagrams and Limits

## 24. Diagrams

### 24.1. Definition

A diagram is the same as *functor.*

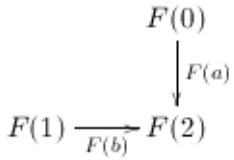
So, further I'll be intermittently using both words. Usually, when people talk about diagrams, they mean a functor from a finite category to a specific category.

### 24.2. Examples

For example, say, we have a category **C**, and two objects, `x0` and `x1`. This is equivalent to having a morphism from a two-point discrete category (**1**+**1**) to category **C**. As a diagram, it is trivial, just two objects.

In a little bit more complicated case we take the following category:


A diagram `F` from such a category to a category `C` looks the same:




Of course, any three objects `X, Y, Z` with two arrows `f: X -> Z` and `g: X -> Z`, looking like


can be interpreted as a diagram from our category with three objects and two arrows: just define `F(0)=X, F(1)=Y, F(2)=Z, F(a)=f, F(b)=g`.

## 25. Limits

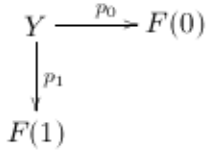
Earlier we already defined an equalizer, a pullback, a Cartesian product, etc. These were particular

examples of limits. Now let's see how we can generalize the idea.

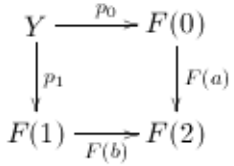
## 25.1. Definition. Cone

Suppose we have a diagram (that is, a functor) `F` from category `A` to category `C`. For an object `y` of category `C` we define a cone from `y` to `F` as a collection of arrows $p_x$: `y -> F(x)` for each object `x` of `A`, such that for any f: $x_0$ -> $x_1$ $p_{x_0} \circ F(f) = p_{x_1}$, that is, the following diagram is commutative:


The object `y`, where all the cone's arrows originate, is called *apex.*
For the two diagrams mentioned above, the cones look like this for **1**+**1**:




and like this

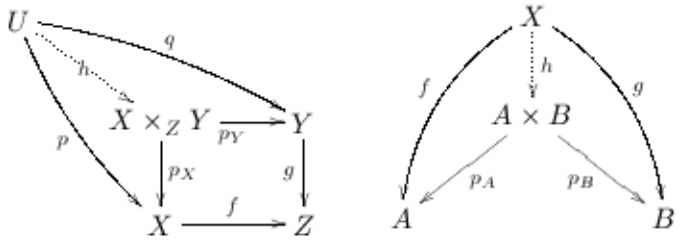



## 25.2. Definition. Limit

There may be more than one cocone from one object to a given diagram; and there is a whole category of all cocones to a given diagram. One of such cones may have a special "universal" feature. It is called limit.

## 25.3. Examples

So we defined a limit as the cone that is nearest to the diagram. Let's recollect the two limits we encountered before: Cartesian product and pullback:




It is clear from the pictures that they are limits for the two diagrams we discussed in section 24.

Now how about primitive cases? A diagram from category **1**? A diagram from category **2**? Try it yourself.

There is an even funnier diagram, from category **0**. What do we have in category **0**? Nothing. So the

cone to a **0**-diagram is just any object. So that the limit of such a diagram is the object to which there is a unique arrow from each object - that is, it is the terminal object of category `C`.

## 25.4. Java Code

Before going ahead and posting pretty clumsy chunks of code implementing this, I recommend you to check out two blog entries where additional infrastructure is described: ***Unlimited Cartesian Product in Java***, part 1 and part 2, and Java Pair: a closer look. The code is, as always, published on the project page.

Additionally, I have added a class `Injection` which extends `Function` and represents an injection; its advantage is that when you apply an `Injection` to a `Set`, you get again a `Set`, since no two elements are mapped to the same one.

First, how do we represent a limit. Since a limit is a special instance of a cone, we need some kind of Java representation for cones. Given a functor `F: X -> Y`, we should have an object `y` in category `Y`, and an arrow $p_x$`: y -> F(x)` for all `x` in category `X`. This requires a `Map<XObjects, YArrows>`, where `XObjects` is the type of objects in category `X`, and `YArrows` is the type of arrows in category `Y`. This map should have all objects of `X` as keys, and for each such object `x` have $p_x$ as a value. Note that it is not enough to specify just the map: the map may be empty (in the case of terminal object defined as a limit).

So we introduce a class `Cone` that is such a pair:

```
public class Cone extends BasePair<YObjects, Map<XObjects, YArrows>> {
 Pair<YObjects, Map<XObjects, YArrows>> delegate;

 public Cone(YObjects y, Map<XObjects, YArrows> map) {
  super(y, map);
 }

  public YObjects apex() {
    return x();
  }

  public YArrows arrowTo(XObjects x) {
    return y().get(x);
  }
}
```

It is an inner class of `Functor`, so we do not have to parameterize it.

Let's throw in a couple of useful methods in this class. The first one checks if this really is a cone, not just a random bundle of arrows:

```
 public boolean isWellFormed() {
  return new Predicate<XArrows>() {
   // take an arrow f: x0 -> x1
   public boolean eval(XArrows f) {
    // an arrow p0: y -> F(x0)
    YArrows yToFx0 = arrowTo(domain().d0(f));
    // an arrow p1: y -> F(x1)
    YArrows yToFx1 = arrowTo(domain().d1(f));
    // F(f)
    YArrows F_f = arrowsMorphism.apply(f);
    // F(f) o p0 must be equal to p1
    return codomain().m(yToFx0, F_f).equals(yToFx1);
   } // this should hold for all arrows f of category X
  }.forall(domain().arrows());
```

```
  }
```

And this one checks if this bundle factors another bundle:

```
  private boolean factorsOnRight(final Cone factored) {
   return new Predicate<YArrows>() {
    public boolean eval(final YArrows h) {
     return new Predicate<XObjects>() {
      public boolean eval(XObjects x) {
       return codomain().m(h, arrowTo(x)).equals(factored.arrowTo(x));
      }
     }.forall(domain().objects());
    }
   }.exists(codomain().arrows(factored.apex(), apex()));
  }
```

What this method says is this: there is an arrow from the other cone's apex to this cone's apex such that it for all `x` in `X` it factors the arrow to `F(x)`.

Now it is easy to define `limit()` method of class `Functor`:

```
 public Cone limit() {
    return new Predicate<Cone>() {
      public boolean eval(Cone candidate) {
        return isLimit(candidate);
      }
    }.findOne(allCones());
  }
```

Here we have two methods to implement: `isLimit()` that checks whether a cone is a limit, and `allCones()` that lists all possible cones to functor `F` in category `Y`.

```
 private boolean isLimit(final Cone candidate) {
    // a candidate is a limit if for each cone to F starting at y
    // the cone is factored through y0; and this is true for all y.
    return new Predicate<Cone>() {
      public boolean eval(Cone anyCone) {
        return candidate.factorsOnRight(anyCone);
      }
    }.forall(allCones());
  }
```

We scan all objects in `Y`, and for each object we scan all possible cones from `y` to `F`, and if each such cone factors through `y0`, then `y0` is a limit. (Why 'a' limit? Because a limit is unique up to an isomorphism.)

Now all that is left is to list all `conesFrom(y)`. To list cones, we need a predicate that check if a given structure is actually a cone:

```
 private Predicate<Cone> isaCone = new Predicate<Cone>() {
    public boolean eval(Cone candidate) {
      return candidate.isWellFormed();
    }
  };
```

So to list all cones means to list all candidate cones and apply the `IsaCone` predicate's filter to produce only those that are cones.

```
Set<Cone> conesFrom(final YObjects y) {
  Set<Set<Pair<XObjects, YArrows>>> homs =
```

```
      new Injection<XObjects, Set<Pair<XObjects, YArrows>>>() {
        // this function builds pairs (x, f:y->F(x)) for all f:y->F(x)) for a given x
        public Set<Pair<XObjects, YArrows>> apply(final XObjects x) {
          return BasePair.<XObjects, YArrows>withLeft(x).
              map(codomain().arrows(y, nodesMorphism.apply(x)));
        }
      }.map(domain().objects());

  Set<? extends Iterable<Pair<XObjects, YArrows>>> productOfHoms = Cartesian.product(homs);
  Set<Set<Pair<XObjects, YArrows>>> setOfSetsOfPairs =
      Set(new IterableToSet<Pair<XObjects, YArrows>>().map(productOfHoms));
  Set<Map<XObjects, YArrows>> allMaps = new PairsToMap<XObjects, YArrows>().map(
      setOfSetsOfPairs);

  Injection<Map<XObjects, YArrows>, Cone> makeCone =
      new Injection<Map<XObjects, YArrows>, Cone>() {
        public Cone apply(Map<XObjects, YArrows> map) {
          return new Cone(y, map);
        }
      };

  return isaCone.filter(makeCone.map(allMaps));
}
```

Not much is left; we only have to enumerate all possible cones, from all objects `y` in `Y` to functor `F`. I am sure not many volunteers will dare to get any meaning of this:

```
Set<Cone> allCones() {
  Injection<YObjects, Set<Cone>> buildAllConesFrom =
      new Injection<YObjects, Set<Cone>>() {
        // For an object y returns all cones from y to F.
        public Set<Cone> apply(final YObjects y) {
          return conesFrom(y);
        }
      };
  return union(buildAllConesFrom.map(codomain().objects()));
}
```

Any suggestion regarding how to simplify this will be greatly appreciated. One solution of course is to rewrite everyting to Haskell or Scala.

And now the unittest:

```
 public void testLimit_cartesianProductActually() {
  Category<Integer, Integer> from = discreteCategory(range(2));
  Category<String, String> to = Category.SQUARE;
  Map<Integer, String> map =
     new HashMap<Integer, String>() {
      {
       put(0, "b");
       put(1, "c");
      }
     };

  Functor<Integer, Integer, String, String> f = Functor(from, to, map, map);
  Pair<String, Map<Integer, String>> limit = f.limit();
  assertNotNull(limit);
  assertEquals("ab", limit.y().get(0));
  assertEquals("ac", limit.y().get(1));
 }
```

This test proves that (well, in our small example) pullback is actually a limit of the diagram that was introduce in the beginning of this chapter, the one with three objects and two arrows.

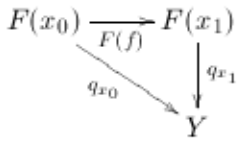
# X. Colimits

## 26. Colimit

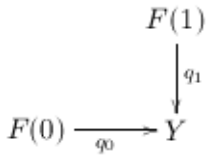
Colimits are the opposite to limits, from categorical point of view; but their real life interpretation is different. Limits are products and subobjects; colimits are unions and factor-objects.

### 26.1. Definition. Cocone

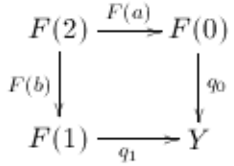
Suppose we have a diagram (that is, a functor) `F` from category `A` to category `C`. For an object `y` of category `C` we define a cocone from `F` to `y` as a collection of arrows $q_x$: `F(x) -> y` for each object `x` of `A`, such that for any `f: x0 -> x1` the following holds: $q_{x_1} \circ F(f) = q_{x_0}$, that is, the following diagram is commutative:




The object `y`, where all the cocone's arrows terminate, is called *apex*, the same as for cones.
For the two diagrams mentioned above, the cocones look like this for **1**+**1**:



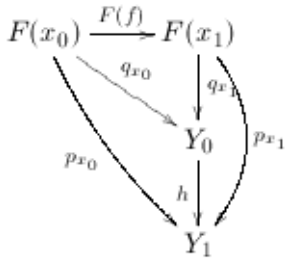

and like this




There may be more than one cocone from a given diagram to a given object; and there is a whole category of all cocones from a given diagram. One of such cocones may have a special "universal" feature. It is called colimit.

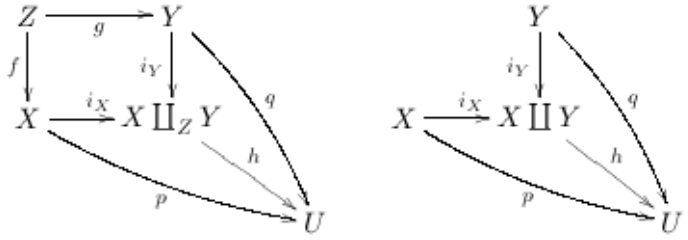
### 26.2. Definition. Colimit

Given a diagram `F: A -> C`, its *colimit* is a cone `F` to $Y_0$ (a collection of $q_x$: `F(x) ->` $Y_0$ for all objects `x` of `A`) such any other cone factors through it. Namely, given any cocone from `F` to $Y_1$ (a collection of $p_x$: `F(x) ->` $Y_1$ for all objects `x` of `A`), there is a unique arrow `h:` $Y_0$ `->` $Y_1$ such that $p_x = h \circ q_x$ for all objects `x` of `A`:




## 26.3. Examples

So we defined a colimit as the cocone that is nearest to the diagram. Let's recollect the two colimits we encountered before: Union and pushout:




It is clear from the pictures that they are colimits for the two diagrams.

Now how about primitive cases? A diagram from category **1**? A diagram from category **2**? Try it yourself.

There is an even funnier diagram, from category **0**. What do we have in category **0**? Nothing. So the cocone from a **0**-diagram is just any object. So that the colimit of such a diagram is the object to which there is a unique arrow to each object - that is, it is the initial object of category `C` (in Sets it is an empty set).

## 26.4. Java Code

This is very similar to the solution for colimits. Start with introducing a class `Cocone` that is such a pair:

```
public class Cocone extends BasePair<YObjects, Map<XObjects, YArrows>> {
   public Cocone(YObjects apex, Map<XObjects, YArrows> map) {
     super(apex, map);
   }

   /**
    * @return the object at which all the arrows of the cocone terminate.
    */
   public YObjects apex() {
     return x();
   }

   /**
    * Given an object x of category X, returns an arrow F(x) -> y of this cocone.
    *
    * @param x an object in X
    * @return the arrow
```

```
     */
    public YArrows arrowFrom(XObjects x) {
      return y().get(x);
    }
  }
```

It is an inner class of `Functor`, so we do not have to parameterize it.

Let's throw in a couple of useful methods in this class. The first one checks if this really is a cocone, not just a random bundle of arrows:

```
  public boolean isWellFormed() {
      return new Predicate<XArrows>() {
        // take an arrow f: x0 -> x1
        public boolean eval(XArrows f) {
          // an arrow q0: F(x0) -> y
          YArrows Fx0ToY = arrowFrom(domain().d0(f));
          // an arrow q1: F(x1) -> y
          YArrows Fx1ToY = arrowFrom(domain().d1(f));
          // F(f)
          YArrows F_f = arrowsMorphism.apply(f);
          // q1 o F(f) must be equal to q0
          try {
            return codomain().m(F_f, Fx1ToY).equals(Fx0ToY);
          } catch (RuntimeException e) {
            throw e;
          }
        } // this should hold for all arrows f of category X
      }.forall(domain().arrows());
    }
```

And this one checks if this bundle factors another bundle:

```
  private boolean factorsOnLeft(final Cocone factored) {
      return new Predicate<YArrows>() {
        public boolean eval(final YArrows h) {
          return new Predicate<XObjects>() {
            public boolean eval(XObjects x) {
              return codomain().m(arrowFrom(x), h).equals(factored.arrowFrom(x));
            }
          }.forall(domain().objects());
        }
      }.exists(codomain().arrows(apex(), factored.apex()));
    }
```

What this method says is this: there is an arrow from this cone's apex to the other cone's apex such that it for all `x` in `X` it factors the arrow to `F(x)`.

Now it is easy to define `colimit()` method of class `Functor`:

```
  public Cocone colimit() {
     return new Predicate<Cocone>() {
       public boolean eval(Cocone candidate) {
         return isColimit(candidate);
       }
     }.findOne(allCocones());
  }
```

Here we have two methods to implement: `isColimit()` that checks whether a cocone is a colimit, and `allCocones()` that lists all possible cocones.
These methods are dual to those for limit calculation, and I won't publish them here.

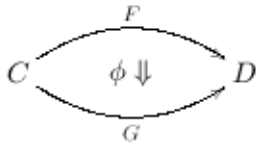
# XI. Natural Transformations

## 27. Natural Transformation

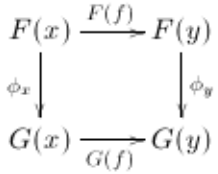
### 27.1. Definition

A natural transformation is a morphism between two functors with the same domain and codomain.

Suppose we have two functors, `F` and `G` from category `C` to category `D`. A natural transformation `φ: F -> G`




is defined by having an arrow $\varphi_x$`: F(x) -> G(x)` for each object `x` in `C`, so that it is compatible with the actions of `F` and `G` on `C`'s arrows, that is, $\varphi_y$ `o F(f) = G(f) o` $\varphi_x$ for each arrow `f: x->y` in `C`:




That's all that is required from natural transformation.

Composing two natural transformations should be obvious: do it per component. Given two categories, `A` and `B`, functors from `A` to `B` form another category, with functors as objects and natural transformations as arrows. Such a category is called $B^A$.

Remember the notions of cone and cocone, introduced in chapters IX and X? They are natural transformations! Take a cone, what is it? It is just a compatible collection of morphism from a given `y` to `F(x)` for each `x`. But we can introduce a functor `Const(y)`, such that each object `x` of `A` maps to `y`, and each arrow of a maps to $1_y$. A cone is a natural transformation `Const(y) -> F`. Similarly, a cocone is a natural transformation `F -> Const(y)`.

### 27.2. Java Code

It is so extremely awkward that I have decided not to continue trying to convert category ides into Java, and will look for a better language.
See:

```
public class NaturalTransformation<
   XObjects, // type of nodes in the first category (like Alksnis?)
   XArrows, // type of arrows in the first category
   YObjects, // type of nodes in the second category
   YArrows   // type of arrows in the second category
   >
```

```
    extends Morphism<
    Functor<XObjects, XArrows, YObjects, YArrows>,
    Functor<XObjects, XArrows, YObjects, YArrows>> {
  private SetMorphism<XObjects, Set<XObjects>, YArrows, Set<YArrows>> Fx_to_Gx;

  public NaturalTransformation(Functor<XObjects, XArrows, YObjects, YArrows> F,
                               Functor<XObjects, XArrows, YObjects, YArrows> G,
                               SetMorphism<XObjects, Set<XObjects>, YArrows, Set<YArrows>>
Fx_to_Gx) {
    super(F, G);
    this.Fx_to_Gx = Fx_to_Gx;
  }

  public static <XObjects, XArrows, YObjects, YArrows>
  NaturalTransformation<
      XObjects,
      XArrows,
      YObjects,
      YArrows
      > unit(final Functor<XObjects, XArrows, YObjects, YArrows> F) {
    return new NaturalTransformation<
        XObjects,
        XArrows,
        YObjects,
        YArrows>(F, F,
        new SetMorphism<XObjects, Set<XObjects>, YArrows, Set<YArrows>>(
            F.domain().objects(), F.codomain().arrows()) {
          public YArrows apply(XObjects x) {
            return F.codomain().unit(F.nodesMorphism.apply(x));
          }
        }
    );
  }
}
```

We probably need to show here what a `Morphism` class is:

Let's throw in a couple of useful methods in this class. The first one checks if this really is a cocone, not just a random bundle of arrows:

```
public abstract class Morphism<X, Y> {
  private String name;
  private X domain;
  private Y codomain;

  /**
   * Base constructor. Takes domain and codomain
   *
   * @param domain domain of this morphism
   * @param codomain codomain of this morphism
   */
  Morphism(X domain, Y codomain) {
    this.domain = domain;
    this.codomain = codomain;
  }

  /**
   * Constructor.
   *
   * @param name name of this morphism
   * @param domain domain of this morphism
   * @param codomain codomain of this morphism
   */
  Morphism(String name, X domain, Y codomain) {
    this(domain, codomain);
    this.name = name;
```

```
  }

  public String name() { return name; }
  public X domain() { return domain; }
  public Y codomain() { return codomain; }

  @Override public String toString() { return name == null ? super.toString() : name; }
}
```

# XII. Category of Java Sets

## 28. What Is It, Category of Java Sets?

In the previous parts we talked about categories in general, but when modeling them in a programming language, we always used a set as a basis: a set of objects, a set of arrows. Not all categories are set-based. Those that are are called small categories. What about a category of all sets? This is still a category. While it is possible, to represent a category of all sets in a programming language, it does not have much practical sense. What does make sense is the attempt to take a category of all sets in a certain language, and look at it from what we already have learned.

So, here I introduce `SETF`, an instance of `Category<Set, Set>` in Java.

This may look a little bit weird: what is the set of its objects? We have to thank the designers of Java collections library for giving us an opportunity to use an uncountable set. The "Set" of objects of `SETF` returns `Integer.MAX_VALUE` as its size, and throws an exception on any attempt to enumerate it or convert to array. The only meaningful method is `contains(Object x)`; since it contains *all* sets, it returns true if the argument `x` is a set.

What remains is to define morphisms and all the regular set constructions as defined in parts V-X of this tutorial.

## 29. TypelessSetMorphism

In previous sections all the activity with categories was strongly typed, via generics. Something like `Morphism<X, Y>: Set<X> -> Set<Y>` was the usual part of discourse. We cannot do it here. The problem is that actually, in "real" Set Theory, there are no types. We can easily build unions of two sets of different nature; in short, sets should not be homogeneous.

So, now we have to abandon almost all the good things we learned about type safety in Java and admit that this is just not the case. Set Theory is typeless, so is Set Category.

That's how we have to introduce an abstract class `TypelessSetMorphism`. It is a typeless (that is, from Objects to Objects) function (see http://vpatryshev.blogspot.com/2008/12/functions.html) that knows its domain and codomain.

The class defines unit morphism, morphism composition, has factories that builds a `TypelessSetMorphism` out of two sets and a function, out of a set and its subset or a set and a predicate (inclusion morphism), out of a set and its factorset, or out of a set and a binary relationship (factorset projection).

```
public static TypelessSetMorphism inclusion(Set subset, Set set) {
  assert set.containsAll(subset) : "It is not an inclusion if it is not a subset.";
  return forFunction(subset, set, Functions.inclusion());
}

public static TypelessSetMorphism inclusion(Set set, Predicate predicate) {
  return inclusion(predicate.filter(set), set);
}

public static <T> TypelessSetMorphism factorset(Set<T> set, Set<Set<T>> factorset) {
  return forFunction(set, factorset, Sets.factorset(set, factorset));
}
```

```
public static <T> TypelessSetMorphism factorset(Set<T> set, BinaryRelationship<T, T> r) {
  Set<Set<T>> factorset = factor(set, r);
  return forFunction(set, factorset, Sets.factorset(set, factorset));
}
```

The biggest method is the one that builds `power(Set a, Set b)` - the set of all `TypelessSetMorphisms` from `a` to `b`:

```
public static Set<TypelessSetMorphism> power(final Set x, final Set y) {
 return new Injection<Map, TypelessSetMorphism>() {

  public TypelessSetMorphism apply(final Map map) {
   return new TypelessSetMorphism(x, y) {
    public Object apply(Object o) {
     return map.get(o);
    }
   };
  }
 }.map(Sets.allMaps(x, y));
}
```

If you are curious how `Sets.allMaps(x, y)` works, just check out the source code.

## 30. SETF

SETF, the category of finite Java sets, is defined like this:

```
public static Category<Set, TypelessSetMorphism> SETF =
  new Category<Set, TypelessSetMorphism>(BigSet.FINITE_SETS) {
   private Set<TypelessSetMorphism> ALL_MORPHISMS =
     new BigSet<TypelessSetMorphism>() {
      public boolean contains(Object f) {
       return f instanceof SetMorphism;
      }
     };

   public TypelessSetMorphism unit(Set x) {
    return TypelessSetMorphism.unitMorphism(x);
   }

   public TypelessSetMorphism m(TypelessSetMorphism f, TypelessSetMorphism g) {
    assert f.codomain() == g.domain() : "Domain and codomain should match";
    return TypelessSetMorphism.compose(f, g);
   }

   public Set<TypelessSetMorphism> arrows() {
    return ALL_MORPHISMS;
   }

   public Set d0(SetMorphism arrow) {
    return (Set) arrow.domain();
   }

   public Set d1(SetMorphism arrow) {
    return (Set) arrow.codomain();
   }

   @Override
   protected void validate() { // it IS a category
   }
```

So far so good, right? No surprises, nothing new. What's harder is implementing all the constructions that we saw in parts V-X. We cannot seriously iterate over all the sets trying to find a pullback, or a union. We have to algorithmically define it right away. But it's okay, they are all constructive. It just took some time to write down the implementations. This implementation will be published will be the next part.

# XIII. Category of Java Sets - Implementation

This part continues Part XII, with the code implementing various limits and colimits in SETF.

## 31. Properties of Morphisms

We cannot inherit much from Category, since in Category all properties are checked by scanning over all morphisms, all objects, or both. But some could be inherited, e.g.

```
@Override
 public boolean isIsomorphism(TypelessSetMorphism arrow) {
   return super.isIsomorphism(arrow);
}
```

Why so? Because the set of arrows from A to B is enumerable, and we can just look up the morphism that is an inverse to the given one, if it exists. Not so with monomorphisms: we are not interested in scanning all the objects of SETF. We could limit ourselves by subobjects of domain, but in a general category there may be no subobjects. So, that's how it looks:

```
 @Override
public boolean isMonomorphism(final TypelessSetMorphism arrow) {
 return new Predicate<Pair<Object, Object>>() {
  public boolean eval(final Pair<Object, Object> p) {
   return equal(arrow.apply(p.x()), arrow.apply(p.y())) ? equal(p.x(), p.y()) : true;
  }
 }.forall(Sets.product(arrow.domain(), arrow.domain()));
}
```

In human languages it means that an arrow is a monomorphism if from `arrow.apply(x)==arrow.apply(y)` it follows that `x == y` for each pair `(x, y)` of elements of the arrow's domain. Kind of obvious, right?

Similarly we can define epimorphims (see the code in the project).

## 32. Equalizer and Coequalizer in SETF

Using the code from `TypelessSetMorphism` class, defining an equalizer is extremely easy. Given a parallel pair of arrows, `f` and `g`, their equalizer is a set of such elements of their (common) domain that `f.apply(x) == g.apply(x)`:

```
 @Override
public TypelessSetMorphism equalizer(
   final TypelessSetMorphism f, final TypelessSetMorphism g) {
 assert f.domain() == g.domain() && f.codomain() == g.codomain();
 return TypelessSetMorphism.inclusion(f.domain(), new Predicate() {
```

```
      public boolean eval(Object x) {
       return equal(f.apply(x), g.apply(x));
      }
     });
    }
```

Coequalizer is a little bit trickier: we have to calculate a factorset by the following binary relationship: `y1 ≡ y2` iff there is an x such that `y1 == f.apply(x)` and `y2 == g.apply(x)`.

```
    @Override
    public TypelessSetMorphism coequalizer(final TypelessSetMorphism f,
       final TypelessSetMorphism g) {
     assertParallelPair(f, g);
     return TypelessSetMorphism.factorset(f.codomain(),new BinaryRelationship<Object,Object>() {
      public boolean eval(final Pair p) {
       Predicate isCommonAncestor = new Predicate() {
        public boolean eval(Object x) {
         return equal(f.apply(x), p.x()) && equal(g.apply(x), p.y());
        }
       };
       return isCommonAncestor.exists(f.domain());
      }
     });
    }
```

Here we define a predicate `isCommonAncestor` that checks if x is such that `y1 == f.apply(x)` and `y2 == g.apply(x)`; then we introduce a `BinaryRelationship` defined by existence of such an `x`, and build a factorset on the transitive symmetric closure of this relationship. If you ever studied Set Theory, it must be obvious. Personally I find it extremely amusing that all this can be expressed in such an awkward language like modern Java.

You are probably curious, what's the deal here about `equal(a, b)`? See, this method is defined in `Category` class, and determines the equality in our egalitarian theory. Since we have `Set`s, and in sets elements by default are equal if their `equals()` returns true, we have this level of equality. Instead, we could use `IdentitySet`s, and so instead of `equals()` could use ==. Not sure how far we can go here, but comparing two stings looking the same could lead to strange results. Let's not go there. I just suggest not to mix two egalitarian theories one discourse.

## 33. Product and Union in SETF

It is more or less obvious how to build a Cartesian product in `SETF`; let's skip it and build a union. Union is a curious thing. You cannot just build a set that contain elements of both: union should be *disjoint*. A union should consist of a set and two injections of the two original sets to the union. Here's how we do it:

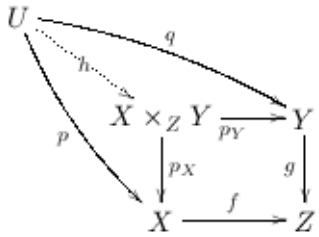
```
    @Override
    public Pair<TypelessSetMorphism, TypelessSetMorphism> union(Set x, Set y) {
     Injection tagX = withLeft("x");
     Injection tagY = withLeft("y");
     Set taggedX = tagX.map(x);
     Set taggedY = tagY.map(y);
     Set unionSet = Sets.union(taggedX, taggedY);
     TypelessSetMorphism ix =
       compose(forFunction(x, taggedX, tagX), TypelessSetMorphism.inclusion(taggedX, unionSet));
     TypelessSetMorphism iy =
```

```
      compose(forFunction(y, taggedY, tagY), TypelessSetMorphism.inclusion(taggedY, unionSet));
    return Pair(ix, iy);
  }
```

The two sets, `taggedX` and `taggedY`, consist of pairs `("x", a)` and `("y", b)` for all `a` in `x` and all `b` in `y`.

## 34. Pullback in SETF

This is more serious. Let me remind you; pullback is defined in SQL as `select (x,y) from X, Y where f(x) = g(y)`, or in the form of diagram:




How does it look in Java?

```
public Pair<TypelessSetMorphism, TypelessSetMorphism> pullback(
  final TypelessSetMorphism f,
  final TypelessSetMorphism g) {
 Pair<TypelessSetMorphism, TypelessSetMorphism> product =
   product(f.domain(), g.domain());

 BinaryRelationship<Object, Object> pairIsGood =
   new BinaryRelationship<Object, Object>() {
  public boolean eval(Pair<Object, Object> pair) {
   return equal(f.apply(pair.x()), g.apply(pair.y()));
  }
 };

 TypelessSetMorphism pullbackToProduct =
   TypelessSetMorphism.inclusion(product.x().domain(), pairIsGood);

 return Pair(
     compose(pullbackToProduct, product.x()),
     compose(pullbackToProduct, product.y()));
}
```

What's going on here? Starting from the end: we return a pair of projections from the pullback to two components. The pullback itself is built as an inclusion to the product of `X` and `Y`; what is included is defined by the `BinaryRelationship` (which is the same as `Predicate<Pair<?, ?>>`) `pairIsGood` - the predicate determines if a pair (x, y) belongs to the pullback, and it does if `f(x) == g(y)`.

Pushout is built somewhat similarly; you can check out the code and see.

## 35. 0, 1, $X^n$

Initial object is an empty set; terminal object is a singleton (a set consisting of an empty set).

I finish this by showing how an object $X^n$ (`X` to the `n`-th power) can be built in `SETF`.

```
@Override
   public Pair<Set, List<TypelessSetMorphism>> degree(final Set x, final int n) {
     final Set<Map> allMaps = Sets.allMaps(range(n), x);
     return Pair(
         (Set) allMaps,
         (List<TypelessSetMorphism>) new AbstractList<TypelessSetMorphism>() {
           public TypelessSetMorphism get(final int i) {
             return new TypelessSetMorphism(allMaps, x) {
               public Object apply(Object map) {
                 return ((Map) map).get(i);
               }
             };
           }

           public int size() {
             return n;
           }
         });
   }
```

Elements of such an object are `List`s of length `n`.

# XIV. Diagrams over Java Sets

Diagrams over Java Sets are as far as I could get implementing categories in Java. Next I'll switch to Scala, go back, and repeat all the previous implementations in that language.

## 36. What's New for Java Sets

As you know from previous chapters, a diagram in category `C` is just a functor from another, probably finite, category to `C`. In this chapter C is the category of Java sets (see parts XII and XII); and, since the default methods do not always work, since we cannot scan over all possible sets, some implementations are different - that's why we need a separate class, `SetDiagram<Objects, Arrows> extends Functor<Objects, Arrows, Set, TypelessSetMorphism>`.

The problem here is that limits and colimits in `Functor` class are calculated and validated by scanning over all the object of domain category. For regular sets we know how to calculate them without global scan, so that' what we do. As a result, the class `SetDiagram` overrides two methods, `limit()` and `colimit()`. I'll talk about these two methods implementations in the following sections.

## 37. Calculating Set Diagram Limit

A set diagram for category `C` consists of sets and typeless set morphisms between them, each set $F_x$ corresponding to an object `x` in category `C` and each morphism $F_a$: $F_x$ -> $F_y$. A cone for such a diagram is an object `A` and a collection of compatible set morphisms $\alpha_x$: $F_x$ -> `A`. A limit is such a cone that every other cone can be factored through it.

So, how do we build a limit? First, if category `C` is discrete (just unit arrows), then the limit is the same as Cartesian product of all $F_x$, $\Pi\{F_x \mid x \in C\}$.

How do we build a product of a set of sets? First, we don't. The problem is that if we have a set of components, their order does not matter. Now the elements of a Cartesian product are usually either pairs, or lists, which do have order. As a result, while $A \times B$ is isomorphic to $B \times A$, they are not the same. So, actually we need to have a list of sets to build a Cartesian product. The fact that these lists are isomorphic, are of second importance. So, to build a limit for a discrete category, we take a list of sets, and build a Cartesian product out of it.

Here's a piece of Java code:

```
List<Objects> listOfObjects = new ArrayList<Objects>(setOfObjects);
List<Set> setsToUse = nodesMorphism.asFunction().map(listOfObjects);
Set<? extends List<Object>> product = Cartesian.product(setsToUse);
```

Now all we need is to have projections from the product to participating sets:

```
Function<Objects, Integer> index = new Function<Objects, Integer>() {
  @Override
  public Integer apply(Objects x) {
    return listOfObjects.indexOf(x);
  }
};
```

```
// This function takes an object and returns a projection set function;
// we have to compose each such projection with the right arrow
// from an object to the image of the object.
final Function<Objects, Function<List<Object>, Object>> projectionForObject =
    index.then(Cartesian.projection());
```

That's all we need for a diagram from a discrete category. A common case is more complicated. In the common case the limit set (that is, cone apex for the limit cone) is a subset of the Cartesian product just built; only those elements are included that are compatible, that is, for `f: x -> y` in category `C` the projection $\alpha_x(p) = F_f(\alpha_y(p))$, where p is the element of the Cartesian product. Projection consists of taking the `x`-th component of the list `p`, so that the condition can be rewritten as $p_x = F_f(p_y)$.

```
final Map<Objects, Set<Arrows>> cobundles =
  domain().op().buildBundles(domain().objects(), arrows);

Predicate<List<Object>> compatibleListsOnly = new Predicate<List<Object>>() {
  @Override
  public boolean eval(final List<Object> point) {
    Predicate<Set<Arrows>> checkCompatibility =
        allArrowsAreCompatibleOnPoint(projectionForObject, point);
    return checkCompatibility.forall(cobundles.values());
  }
};
return compatibleListsOnly.filter(product);
```

This is the essence of the code of methods `calculateLimitApex()` and `limit()` of class `SetDiagram`. The actual implementation is a little bit trickier, since in many cases we can omit some of the object of category `C`. For instance, look at category **2**. The diagram over **2** looks like this: `A -> B`, where `A` and `B` are two arbitrary sets. Do we need `B` to build a limit? No; the limit for this diagram is just `A`. So `B` can be omitted. In general case, all objects that do not have arrows back to the other objects pointing at them can be omitted.

The unittest class for `SetDiagram` class contains a bunch of specific examples.

## 38. Calculating Set Diagram Colimit

Theoretically speaking, colimit is just a limit but calculated in an op-category. No, I do not advise anybody to go ahead and start manipulating $\texttt{Set}^{op}$. Wikipedia writes this about $\texttt{Set}^{op}$: "The category $\mathbf{Set}^{op}$ can be embedded into **Rel** by representing each set as itself and each function $f\colon X \to Y$ as the relation from $Y$ to $X$ formed as the set of pairs $(f(x),x)$ for all $x \in X$; hence $\mathbf{Set}^{op}$ is concretizable. The forgetful functor which arises in this way is the contravariant powerset functor $\mathbf{Set}^{op} \to \mathbf{Set}$."

So again, we have to calculate colimit "manually".

Let's start with a discrete category `C`. In this case colimit is just a union of all sets involved in the diagram: $\cup\{Fx: x \in C\}$.

It is nice when all $F_x$ are distinct and disjoint; but we cannot count on it, so that just building a union of all participating sets is not an option. The trick is to tag all elements. For an `i`-th component of the union, $A_i$, we build `{Pair(i, x) | x∈Ai}`. `Ai` is isomorphic to such set, and $A_i$ and $A_j$ are disjoint for `i != j`.

Here's the Java code:

```
final List<Objects> listOfObjects = new ArrayList<Objects>(objects);
// Here we have a non-repeating collection of sets to use for building a union
List<Set> setsToUse = nodesMorphism.asFunction().map(listOfObjects);
List<Set<Object>> setsToJoin = new Id().map(setsToUse);
```

```
DisjointUnion<Object> du = new DisjointUnion<Object>(setsToJoin);
```

We need to convert `Set` to `Set<Object>`, for `DisjointUnion` to like it (that's what `Id()` trick does); DisjointUnion is a nice wrapper for calculating disjoint union of sets and their inclusions to the union. Again, we had to build a list, to keep a specific order and to provide component tags.

That will be it for a discrete category, but for an arbitrary category things are way trickier. We will need to calculate a factorset of the disjoint union. Similar to what we did for calculating limits, except that two elements `y1` and `y2` of the union are declared equivalent if there is a set `A` in the diagram and an element `x ∈ A`, and two maps in the diagram, f and `g`, such that `y1 = f(x)` and `y2 = g(x)`. We do transitive closure of this relationship, so that we can calculate the factorset. The problem is that it takes pretty heavy calculations to check each pair; and it means that I've decided to forget about lazy functionalness and just do straight calculations.

An earlier introduced class `FactorSet` has a convenient method called `merge()`. It helps build a factorset iteratively. In the beginning, a factorset is the same as the source set `X`, but then we declare two elements `x1 ∈ X, x2 ∈ X` equivalent, thus merging their equivalence classes if they were distinct.

Here's how the main loop for calculating our colimit object looks like:

```
final FactorSet factorset = new FactorSet(Pair(du.unionSet(), du.injections()).x());
// have to factor the union by the equivalence relationship caused
// by two morphisms mapping the same element to two possibly different.
for (Objects o : domain().objects()) {
  Set from = nodesMorphism.apply(o);
  TypelessSetMorphism f = null;

  for (Arrows a : bundles.get(o)) {
    TypelessSetMorphism aAsMorphism = arrowsMorphism.apply(a);
    TypelessSetMorphism toUnion =
        TypelessSetMorphism.forFunction(
            aAsMorphism.codomain(),
            du.unionSet(),
            objectToInjection.apply(domain().d1(a)));
    TypelessSetMorphism g = aAsMorphism.then(toUnion);
    canonicalTSMPerObject.put(o, g);
    if (f != null) {
      for (Object x : from) {
        factorset.merge(f.apply(x), g.apply(x));
      }
    }
    f = g;
  }
}
final TypelessSetMorphism factorMorphism = TypelessSetMorphism.forFactorset(factorset);
```

Here `du` is our disjoint union; `objectToInjection` returns, for an object `x`, an injection $F_x$ `-> disjoinunion`, and `factorset` is what we are building.

The rest is more or less technicalities; it is worth noting just that we do not need to use all the objects of the diagram to build the colimit; similar technique is used to limit the set of participating objects as was done in `limit()`.

Check out the code and the examples of colimit calculations in the unittest. For instance, this unittest checks the pushout diagram:

```
public void testColimit_pushoutActually() {
  SetDiagram<String, String> diagram =
```

```
        buildDiagram(Categories.PUSHOUT, buildObjects("123", "12345", "01234"));
    Functor<String, String, Set, TypelessSetMorphism>.Cocone actual = diagram.colimit();
    Set actualSet = actual.apex();
    Set expectedSet = Set(
        Set(Pair(1, "c0")),
        Set(Pair(0, "b1"), Pair(1, "c1")),
        Set(Pair(0, "b2"), Pair(1, "c2")),
        Set(Pair(0, "b3"), Pair(1, "c3")),
        Set(Pair(1, "c4")),
        Set(Pair(0, "b4")),
        Set(Pair(0, "b5")));
    assertEquals(expectedSet, actualSet);
    assertEquals(
        Map(
            array("b1", "b2", "b3", "b4", "b5"),
            array(Set(Pair(0, "b1"), Pair(1, "c1")), Set(Pair(0, "b2"), Pair(1, "c2")), Set(Pair(0, "b3
"),Pair(1, "c3")), Set(Pair(0, "b4")), Set(Pair(0, "b5")))),
        actual.arrowFrom("b").asMap());
}
```

That's it for Java.