Implementing the Categorical Approach to Graph Transformations With Java

Hans J. Schneider
Universität Erlangen-Nürnberg – Department Informatik (Lehrstuhl 2)
Martensstraße 3, 91058 Erlangen, Germany
schneider@informatik.uni-erlangen.de

Abstract. The categorical approach to graph transformations is well-suited to study the generic concepts of modern programming languages. Here, we present an implementation of some categorical definitions and constructions in Java, and we show how this language supports the genericity of the categorical approach. We apply the constructions to the category of sets and the category of graphs.

1 Introduction

The categorical approach to graph transformations [3] is highly generic: All the proofs and constructions are valid for various types of graphs (unlabeled, node labeled, edge labeled, different ways of labeling, etc.). Since modern programming languages like Java support generic concepts, it is a promising idea to present the categorical approach to graph transformations in such a language. We can implement the general definitions and constructions without referring to special versions of graphs. If we subsequently consider a special type of graphs, we have to define only some basic operations in detail. All the other stuff is inherited from the generic modules, i.e., it must be implemented only once. For reason of space, the present version of the paper is restricted to the basic categorical notions, and does not yet include derivability.

A related concept has been published by Burstall and Rydeheard using ML [1]. Their approach is mainly based on polymorphism, whereas we can take advantage of Java’s class concept.

2 The Generic Modules

We start with summarizing some basic definitions and constructions of category theory and translate them into Java straightforwardly. The presentation is based on our textbook [3, Chapter 2], where the reader can find more detailed explanations and examples.
2.1 Category

The package category includes all the generic definitions and constructions. We start with the definition of a category. From the programming point of view, it is more convenient to use the version that is introduced as an exercise in the textbook:

**Definition 1 (Category):** A category is a quintuple

\[ C = (\text{Obj}_C, \text{Mor}_C, \text{dom}_C, \text{codom}_C, \cdot_C) \]

where the following conditions hold:

1. \( \text{dom}_C, \text{codom}_C : \text{Mor}_C \to \text{Obj}_C \)
2. \( \cdot_C : \text{Mor}_C \times \text{Mor}_C \to \text{Mor}_C \) is a partial function with the following properties:
   - \( g \cdot_C f \) exists if and only if \( \text{codom}_C(f) = \text{dom}_C(g) \).
   - \( \text{dom}_C(g \cdot_C f) = \text{dom}_C(f) \land \text{dom}_C(g \cdot_C f) = \text{dom}_C(g) \).
   - If in \( (h \cdot_C g) \cdot_C f = h \cdot_C (g \cdot_C f) \), one side of the equation exists, the other also exists and both are equal.
   - For each object \( A \in \text{Obj}_C \), there exists a morphism \( \text{id}_A \in \text{Mor}_C \) such that \( (\forall A, B \in \text{Obj}_C)(\forall f : A \to B) (\text{id}_B \cdot_C f = f = f \cdot_C \text{id}_A) \).

First, we have to decide on whether to use interfaces or abstract classes in implementing objects, morphisms, and categories. Using interfaces, we can take advantage of multiple inheritance, but they do not allow implementation of default methods. The categories of interest have both limits and colimits. Therefore, we prefer an interface in this case, since putting all the constructions into one class makes it very large and difficult to comprehend. Furthermore, that solution would contradict the principle of separation of concerns.

Implementing categorical constructions in a generic way, i.e., without referring to a concrete category, we need methods that construct objects and morphisms. Therefore, the interface must ensure that concrete categories define these methods:

```java
public interface Cat {
    public Obj makeObj(Object o);
    public Mor makeMor(Obj from, Arrow f, Obj to);
}
```

Of course, the implementations of these methods depend on the category under consideration.

We must distinguish the categorical objects from Java objects, since a categorical object has special properties, e.g., there is an identity morphism related to it. On the other hand, we do not want to restrict the programming language

---

1 Later on, we usually omit the index \( C \), since the category under consideration is obvious.
constructs that may be used as categorical objects: Any element of the Java class `Object` is allowed to be converted into a categorical object.

A morphism connects two categorical objects by an arrow. Considering different categories, we find a large variety of arrows. Therefore, we introduce only an interface ensuring that an appropriate check for equality is provided by the implementation:

```java
public interface Arrow {
    public boolean equals(Arrow f);
}
```

The class `Obj` describes the methods common to all categorical objects. We provide each object with the category it belongs to and with the identity morphism associated with the object according to Definition 1:

```java
public abstract class Obj {
    protected Cat cat;
    protected Mor idSaved;

    public Cat isIn() { return cat; }

    public Mor id() {
        if (idSaved == null) { idSaved = computeId(); }
        return idSaved;
    }
}
```

The identity morphism is not computed until it is needed. Then, it is saved for further usage:

```java
public boolean equals(Obj o) {
    if (this == o) { return true; }
    else if (!o.isIn().equals(cat)) {
        throw new CatError('Obj.equals: not the same category');
    } else { return equalsSemant(o); }
}
```

We have to check equality semantically in the category under consideration:

```java
protected abstract boolean equalsSemant(Obj o);}
```

Although multiple inheritance occurs in the hierarchy of morphisms, we prefer an abstract class in this case. The reason is that we can take advantage of using default methods. The abstract class `Mor` describes the methods common to all morphisms.
the morphisms. First, we define methods allowing applications to access the local fields from outside:

```java
public abstract class Mor {

    protected Cat cat;
    protected Obj dom, codom;
    protected Arrow arrow;

    public Cat isIn() { return cat; }
    public Obj dom() { return dom; }
    public Obj codom() { return codom; }
    public Arrow arrow() { return arrow; }

    The main operation we can perform on morphisms is to compose them. We divide this operation into two methods: The first checks applicability, and it is a method common to all categories, whereas the second implements the details. Of course, the second must be abstract:

```java
    public Mor compose(Mor m) {
        if ( !this.dom.equals(m.codom) )
            throw new CatError
            ("Mor.compose: Morphisms don't match");
        return this.composeWithoutTest(m);
    }
    protected abstract Mor composeWithoutTest( Mor m );
```

Separating these two methods has some advantages, such as omitting superfluous tests in categorical constructions where the morphisms usually fit together by definition. Defined as a protected method, however, it can not be used from outside.

Please note that the Java syntax `this.compose(m)` immediately corresponds to the categorical notation `this · m`.

Of course, we also need a method to test morphisms for equality. Since equality tests almost always have to do with commutative diagrams, we use the term `commutes` instead of `equals`:

```java
    public boolean commutes (Mor m) {
        if ( this == m ) { return true; }
        else return this.dom.equals(m.dom) &&
            this.codom.equals(m.codom) &&
            this.arrow.equals(m.arrow);
    }
```

We do not check that both morphisms belong to the same category, since it is done twice in comparing the domains and the codomains!

Furthermore, we need some more methods such as testing for special properties (`isMonomorphic` etc.) and printing, but we omit them here for reason of space.
2.2 Categories With Colimits

An advantage of category theory is that it provides us with general constructions applicable in different areas, e.g., we can construct all colimits in any category that has initial object, coproduct, and coequalizer, and this construction does not depend on the special category. We implement the colimits as abstract classes.\footnote{At first glance, it makes sense to collect all the classes and interfaces concerning colimits in a package \texttt{category.colimits}, but in that case, we can not immediately access the protected fields of the classes defined in the package \texttt{category}.}

We can define the interfaces by simply looking at the corresponding definitions. We start with the binary coproduct:

\textbf{Definition 2 (Coproduct):} The \textit{coproduct} of a pair of objects \((O_1, O_2)\) is a triple \((m_1, m_2, O)\) where \(O\) is an object and \(m_1 : O_1 \to O\) and \(m_2 : O_2 \to O\) are morphisms such that

\[
\forall O' \left( \forall f_1 : O_1 \to O' \right) \left( \forall f_2 : O_2 \to O' \right) \left( \exists! u : O \to O' \right) (f_1 = u \cdot m_1 \land f_2 = u \cdot m_2)
\]

A category is said to \textit{have coproducts} if there exists a coproduct for every pair of objects in \(\text{Obj}_C\).

The condition can be interpreted graphically:

\[
\begin{array}{c}
O_1 \\
\Downarrow^{m_1} \searrow_{f_1} \\
O \quad = \quad O' \\
\swarrow_{u} \nearrow^{f_2} \\
O_2 \\
\Uparrow^{m_2}
\end{array}
\]

We immediately translate this definition into an abstract Java class:

```java
public abstract class BinCoproduct {
    protected Cat cat;
    protected Mor mor1, mor2;
    protected Obj cp;
}
```

Of course, we can also save the objects we start from; but usually, they are already known to the program calling the construction of a pushout, and therefore, we omit them.\footnote{The only exception we have found up to now is constructing the coproduct complement. In this case, however, the unknown object can be found by looking for the domain of the complement.}

We provide this abstract class with methods allowing access to its members, i.e., to the components a coproduct is constructed of:
The universal property depends on the category under consideration and therefore, makes the class an abstract one:

```java
public abstract Mor univ (Mor f1, Mor f2);
}
```

Whereas the coproduct construction starts from two objects, the next example of a colimit starts from two parallel morphisms:

**Definition 3 (Coequalizer):** A morphism \( m : O_2 \to O \) is called the coequalizer of a pair of parallel morphisms \( m_1, m_2 : O_1 \to O_2 \) if and only if the following conditions hold:

(a) \( m \cdot m_1 = m \cdot m_2 \)

(b) \( \forall q : O_2 \to O' \)(\( q \cdot m_1 = q \cdot m_2 \Rightarrow (\exists ! u : O \to O')(q = u \cdot m) \))

A category is said to have coequalizers if every pair of parallel morphisms has a coequalizer.

Again, we illustrate the factorization graphically:

\[
\begin{array}{ccc}
O_1 & \xrightarrow{m_1} & O_2 \\
\downarrow{m_2} & & \downarrow{m} \\
O & \xleftarrow{q} & O' \\
& \uparrow{u} & \\
& & O'
\end{array}
\]

As before, the Java code is straight-forward:

```java
public abstract class Coequalizer {
    protected Cat cat;
    protected Mor mor1, mor2, mor;
    protected Obj obj1, obj2, obj;

    public Cat cat() { return cat; }
    public Obj obj() { return obj; }
    public Mor mor() { return mor; }
    public abstract Mor univ (Mor q);
}
```

The last and the simplest of the basic colimit constructions is the initial object:

**Definition 4 (Initial object):** An object \( O \) is called the initial object of category \( C \) provided that for all objects \( O' \in \text{Obj}_C \), there is exactly one morphism \( O \to O' \).
public abstract class InitialObj {
    protected Cat cat;
    protected Obj init;

    public Obj obj () { return init; }
    public abstract Mor univ ( Obj o );
}

These basic colimits must be implemented by each category in a specific way, i.e., we have to implement a constructor and to complete the method univ. Then, the other colimits can be constructed in a generic way: A category that has initial object, coproduct, and coequalizer has all finite colimits. Therefore, we define an interface CatWithColimits assuming the existence of these basic colimits and providing methods to construct the other colimits such as the pushout:

public interface CatWithColimits extends Cat {
    public InitialObj makeInitialObj ();
    public BinCoproduct makeBinCoproduct (Obj o1, Obj o2 );
    public Coequalizer makeCoequalizer (Mor g, Mor h );

    public Pushout pushout ( Mor f, Mor g );
}

The dual constructions lead to categories with limits, and of course, there are categories having colimits as well as limits. This is the reason why we have chosen an interface.

To illustrate constructing arbitrary finite colimits from the basic ones, we have equipped this interface with a method constructing pushout diagrams. The pushout construction starts from two morphisms with the same domain (interface object), but different codomains:

Definition 5 (Pushout): A commutative square \( \bar{m}_2 \cdot m_1 = \bar{m}_1 \cdot m_2 \) of morphisms is called the pushout diagram of \((m_1, m_2)\) if and only if the following condition holds:

\[
(\forall O')(\forall f_1 : O_1 \to O') (\forall f_2 : O_2 \to O')
(\forall f : O \to O')(f_1 \cdot m_1 = f_2 \cdot m_2 \Rightarrow \exists u : O \to O'(f_1 = u \cdot m_2 \land f_2 = u \cdot m_1))
\]

\(O\) is called the pushout object. A category is said to have pushouts if there exists a pushout diagram for every pair of morphisms with common domain.

Again, we can translate the diagram straightforwardly:

public abstract class Pushout {
    protected Mor mor1, mor2, mor1b, mor2b;
    protected Obj objI, obj1, obj2, objPO;

6 The details of this method are discussed in the next subsection.
Diagram illustrating Def. 5

\[
\begin{array}{c}
I \\
\downarrow m_1 \\
O_1 \\
\downarrow m_2 \\
O_2 \\
\downarrow m_1 \\
O = f_1 \\
\downarrow f_2 \\
O' \\
\end{array}
\]

The field labels as well as the method identifiers are chosen according to the diagram, e.g., \textit{mor1} denotes the morphism \(m_1\), and \textit{mor1b} denotes \(\bar{m}_1\), the letter \(b\) indicating the bar.

It seems unnecessary to provide the class with access to all the components of the diagram, since an application constructing the pushout diagram from the morphisms \(m_1\) and \(m_2\) evidently knows the objects \(I, O_1, O_2\) as well as the given morphisms. Implementing derivation steps, however, we also use the pushout complement constructing \(m_2\) and \(O_2\) as the results. In that case, it may be more convenient to have access to these components.

### 2.3 General Construction of Pushouts

Whereas \texttt{InitialObj}, \texttt{BinCoproduct}, and \texttt{Coequalizer} must be implemented for each category, we can give a default implementation to construct pushouts. Since \texttt{CatWithColimits} is an interface, we cannot implement this construction as a default method. Therefore, we encapsulate it into a factory class \texttt{PushoutCreator}. Each category, e.g., the category of sets, must define a local object of this class to access the method \texttt{pushout}.

Objects of this factory class make available the methods we need to construct pushouts.
public class PushoutCreator {

    private Cat cat;

    public PushoutCreator ( Cat cat ) {
        this.cat = cat;
    }

    The method implementing the standard construction first tests that the morphisms start from the same object:
    public Pushout pushout ( Mor mor1, Mor mor2 ) {
        if ( mor1.dom() != mor2.dom() )
            throw new CatError
                ("Pushout: No common source object");
        return new CompPushout(mor1,mor2);
    }

    Theorem 6 (Canonical construction of pushouts): A category with co-products and coequalizers has pushouts, too.

    The proof is constructive and can easily be translated into Java code along Figure 6a. (The details of the proof can be found in [3].)
The constructor of the class `CompPushout` implements the diagram and yields the pushout object:

```java
private class CompPushout extends Pushout {
    private BinCoproduct cp;
    private Coequalizer ce;
}
```

We first copy some of the objects and morphisms that can be derived from the parameters:

```java
public CompPushout ( Mor mor1, Mor mor2 ) {
    this.mor1 = mor1; this.mor2 = mor2;
    objI = mor1.dom();
    obj1 = mor1.codom(); obj2 = mor2.codom();
}
```

As the diagram shows, we need two intermediate constructions, a binary coproduct and a coequalizer:

```java
  cp = ((CatWithBinCoproduct) cat).makeBinCoproduct
      (mor1.codom(),mor2.codom());
  ce = ((CatWithCoequalizer) cat).makeCoequalizer
      (cp.mor1().compose(mor1),
       cp.mor2().compose(mor2));
```

Then, we copy the constructed object and the new morphisms into the local fields:

```java
  objPO = ce.obj();
  mor2b = ce.mor().compose(cp.mor1());
  mor1b = ce.mor().compose(cp.mor2());
```

Finally, we provide the object with a method applying the universal property. Figures 6b and 6c illustrate how to construct $u$ from two morphisms $f_1$ and $f_2$ with $f_2 \cdot m_2 = f_1 \cdot m_1$ by applying the universal properties of the coproduct and the coequalizer.

```java
public Mor univ ( Mor f1, Mor f2 ) {
    return ce.univ(cp.univ(f1,f2));
}
```

Of course, we omit the proof showing that this construction satisfies the universal property.
2.4 Composing Pushouts

Two pushout diagrams can be composed and the result is a pushout diagram again:

**Lemma 7 (Composing pushouts):** Consider the following commutative diagram:

\[
\begin{array}{ccc}
A & \xrightarrow{m_{11}} & B \\
\downarrow{m_{12}} & & \downarrow{m_{21}} \\
C & \xrightarrow{m_{11}} & D
\end{array}
\]

If both the left-hand subdiagram \( \overline{m_{12}} \cdot m_{11} = \overline{m_{11}} \cdot m_{12} \) and the right-hand subdiagram \( \overline{m_{22}} \cdot m_{21} = \overline{m_{21}} \cdot m_{12} \) are pushout diagrams, then so is the outer diagram \( \overline{m_{22}} \cdot m_{21} \cdot m_{11} = \overline{m_{21}} \cdot \overline{m_{11}} \cdot m_{12} \).

To implement this lemma, we simply add another method to the already defined class `PushoutCreator`. Java allows us to use the same identifier since the methods can be distinguished by the types of the parameters:

```java
public Pushout pushout ( Pushout po1, Pushout po2 ) {
    if ( !po1.mor2b().equals(po2.mor2()) ) {
        throw new CatError
            ("Composing pushouts: don’t match");
    }
    return new ComposingPushouts(po1, po2);
}
```

Of course, we can put together two pushout diagrams only if the right-hand side of the first and the left-hand side of the second are equal.\(^7\) If this condition is satisfied, we can copy the components of the resulting pushout diagram immediately from the given ones:

```java
private class ComposingPushouts extends Pushout {
    private Pushout po1, po2;

    public ComposingPushouts ( Pushout po1, Pushout po2 ) {
        this.po1 = po1; this.po2 = po2;
        this.objI = po1.objI();
        this.obj1 = po2.obj1();
        this.obj2 = po1.obj2();
        this.objPO = po2.obj();
        this.mor1 = po2.mor1().composeWithoutTest(po1.mor1());
        this.mor2 = po1.mor2();
        this.mor2b = po2.mor2b();
        this.mor1b = po2.mor1b().composeWithoutTest(po1.mor1b());
    }
}
```

\(^7\) More precisely, the morphisms must be identical up to isomorphism. But, checking this is too expensive to be implemented here.
It is sufficient to use `composeWithoutTest` instead of `compose`, since the diagram ensures the correctness.

The universal property can be easily implemented considering the diagram used in the proof. First, we construct $h$ using the pushout property of the left-hand diagram. Then, we get $u$ by the pushout property of the right-hand diagram:

```java
public Mor univ ( Mor f1, Mor f2 ) {
    Mor h = po1.univ(f1.compose(po2.mor1()), f2);
    return po2.univ(f1, h);
}
```

3 The Category Set

3.1 Definition

Since there are a lot of ways to implement sets, we restrict the class `CatFinSet` to represent the common properties. We arrange this definition together with all the other classes we need in a package `category.catfinset`:

```java
public abstract class CatFinSet implements CatWithColimits {
    private PushoutCreator po;
    public CatFinSet () {
        po = new PushoutCreator(this);
    }
}
```

As we have already mentioned, each category that has pushouts must have a local pushout creator that implements the general construction of pushouts. If an application wants to construct a pushout diagram, `CatFinSet` forwards this message to its local pushout creator:
public Pushout pushout (Mor mor1, Mor mor2) {
    return po.pushout(mor1,mor2);
}

The basic colimits must be implemented explicitly, and these implementations must fit in with the definitions given in the interfaces for categories:
public InitialObj makeInitialObj() {
    return new InitialObjFinSet(this);
}

public BinCoproduct makeBinCoproduct(Obj o1, Obj o2) {
    return new BinCoproductFinSet((ObjFinSet) o1,
                                (ObjFinSet) o2);
}

public Coequalizer makeCoequalizer(Mor m1, Mor m2) {
    return new CoequalizerFinSet((MorFinSet) m1,
                                (MorFinSet) m2);
}

Only the first of these methods must be explicitly provided with the parameter this; in the other methods, the objects and morphisms make it available.

In the interface Cat, we have introduced abstract methods to create objects and morphisms. We have to adjust these methods to the properties of Set. i.e., the objects are sets and the arrow is a map from one set to the other:
public ObjFinSet makeObj(Object o) {
    return new ObjFinSet(this, (FinSet) o);
}

public MorFinSet makeMor(Obj from, Arrow f, Obj to) {
    return new MorFinSet(this, (ObjFinSet) from, (FinSetMap) f, (ObjFinSet) to);
}

protected MorFinSet makeMor
    (ObjFinSet from, FinSetMap f, ObjFinSet to) {
    return new MorFinSet(this, from, f, to);
}

The last method is for efficiency. It can be used within the package catfinset to avoid unnecessary cast operations.

The class CatFinSet is abstract to allow implementing sets in different ways, e.g., as hashed sets, as linked lists, or by numbering the elements. Here, we only ensure that the definitions are given elsewhere:

8 Of course, it makes no sense to use any object that is not a finite set, but we must implement the method defined in the interface Cat.
9 An efficient implementation is numbering the elements from 1 to n, where n is the cardinality of the set, a map is simply an array of integers. If we, however, use other notations, the examples can be read more intuitively.
public abstract FinSet makeSetImpl();
/* defines a concrete implementation of sets */

public abstract FinSetMap makeMapImpl();
/* defines a concrete implementation of maps */

Our implementation also includes some methods to handle monomorphisms
and epimorphisms; here, we omit these methods for reason of space.

3.2 Objects and Morphisms

We must distinguish between a usual set and a set as a categorical object:

```java
public class ObjFinSet extends Obj {

    private FinSet set;

    Of course, the categorical object has a local field referring to the underlying set.
The constructor makes a usual set an object of the category $\text{Set}$:

```java
public ObjFinSet(CatFinSet cat, FinSet set) {
    this.cat = cat;
    this.set = set;
}
```

Conversely, we can forget the categorical properties:

```java
public FinSet asSet() {
    return set;
}
```

The sematical equality simply is the equality of the underlying sets:

```java
protected boolean equalsSemant(Obj s) {
    return set.equals(((ObjFinSet) s).set);
}
```

Finally, we have to provide the categorical objects with a method how to
construct the object’s identity morphism. In the case of sets, this is the function
mapping each element onto itself.\footnote{\text{We allow sets to contain any Java object.}}

```java
protected Mor computeId() {
    FinSetMap fct = ((CatFinSet)cat).makeMapImpl();
    Iterator iter = set.iterator();
    Object element;
    while ( iter.hasNext() ) {
        element = iter.next();
        fct.add(element, element);
    }
    return ((CatFinSet)cat).makeMor(this, fct, this);
}
```
Morphisms of $\mathsf{Set}$ are constructed from two objects and a function between the underlying sets. To access these components efficiently, we provide the objects of $\text{MorFinSet}$ with appropriate local fields:

```java
public class MorFinSet extends Mor {
    protected FinSetMap fct;
}
```

The constructor copies its arguments into the local fields, and it must check that the mapping assigns a value in the set of the target object to each argument in the set of the source object:

```java
public MorFinSet(CatFinSet cat, ObjFinSet from, FinSetMap fct, ObjFinSet to) {
    this.cat = cat;
    this.from = from;
    this.to = to;
    this.fct = fct; this.arrow = fct;
}
```

In $\mathsf{Set}$, the arrows are set mappings. For reason of efficiency, we store them locally in this way; but globally, they are of type $\text{Arrow}$. This means that we have to define $\text{FinSetMap}$ as an implementation of the interface $\text{Arrow}$.

```java
if ( !fct.domain().equals(from.asSet()) ) {
    throw new CatError("MorFinSet: Wrong domain");
}
if ( !fct.range().subsetOf(to.asSet()) ) {
    throw new CatError("MorFinSet: Wrong codomain");
}
```

Finally, we have to implement how to compose morphisms in $\mathsf{Set}$:

```java
protected Mor composeWithoutTest(Mor m) {
    FinSetMap fct1 = ((MorFinSet) m).fct;
    FinSetMap fct2 = this.fct;
    FinSetMap fct = ((CatFinSet)cat).makeMapImpl();
    Iterator iter = ((ObjFinSet)m.dom()).asSet().iterator();
    Object element;
    while ( iter.hasNext() ) {
        element = iter.next();
        fct.add(element,
                fct2.getValue(fct1.getValue(element)));
    }
    return ((CatFinSet)cat).makeMor
                ((ObjFinSet) m.dom(), fct, this.to);
}
```

The method does not check that the domain of the morphism is equal to the codomain of the argument $m$. This is already done by the method $\text{compose}$ in the superclass.
3.3 Initial Object and Coproduct

We can construct all the colimits applying the generic methods if we have implemented the basic colimits, i.e., the initial object, the coproduct, and the coequalizer. Thus, we have to implement only these abstract classes. In the case of the initial object, we add a field allowing local access to the implementation of the object as a set:

```java
public class InitialObjFinSet extends InitialObj {
    private FinSet initImpl;
}
```

**Lemma 8 (Initial object in Set):** The initial object in Set is the empty set.

This lemma is easily translated into the constructor of the class:

```java
public InitialObjFinSet (CatFinSet cat) {
    this.cat = cat;
    initImpl = cat.makeSetImpl();
    init = new ObjFinSet(cat, initImpl);
}
```

The universal property returns an empty map from the empty set to a given object o:

```java
public Mor univ (Obj o) {
    return cat.makeMor(init,
        ((CatFinSet) cat).makeMapImpl(),
        (ObjFinSet) o);
}
```

The implementation of the coproduct is analogous:

```java
public class BinCoproductFinSet extends BinCoproduct {
    private CatFinSet c;
    private FinSet s1, s2, cpset;
    private FinSetMap m1map, m2map;
}
```

For reason of efficiency, we have added some fields allowing local access to the category the coproduct belongs to without casts and to the underlying sets and set mappings. The constructor first copies references to the underlying sets:

```java
public BinCoproductFinSet (ObjFinSet o1, ObjFinSet o2) {
    this.cat = o1.isIn();
    c = (CatFinSet) cat;
    s1 = o1.asSet();
    s2 = o2.asSet();
    cpset = c.makeSetImpl();
    m1map = c.makeMapImpl();
    m2map = c.makeMapImpl();
}
```
Lemma 9 (Coproduct in $\textbf{Set}$): The coproduct in $\textbf{Set}$ is the disjoint union together with the natural injections.

Since we can not assume that the given sets are disjoint, we apply a trick: We represent each element of the given sets by a string using the $\text{toString}$-method of its class and prefix the character 1 to this string in the case of the first object, and the character 2 in the case of the second. Then, we can simply put the elements together:

```java
Iterator iter = s1.iterator();
while ( iter.hasNext() ) {
    String s = (String) iter.next();
    String t = Integer.toString(1) + s;
    cpset.add(t);
    m1map.add(s,t);
}
iter = s2.iterator();
while ( iter.hasNext() ) {
    String s = (String) iter.next();
    String t = Integer.toString(2) + s;
    cpset.add(t);
    m2map.add(s,t);
}
```

Please note that we have constructed the union and the natural injections mapping the given sets into it in the same step. Finally, we transform them into morphisms:

```java
cp = new ObjFinSet(c, cpset);
mor1 = new MorFinSet(c, o1, m1map, (ObjFinSet) cp);
mor2 = new MorFinSet(c, o2, m2map, (ObjFinSet) cp);
}
```

To implement the universal property, we consider two arbitrary mappings $f'_1: O_1 \rightarrow C'$ and $f'_2: O_2 \rightarrow C'$ as given in the figure of Def. 2:

```java
public Mor univ ( Mor f1, Mor f2 ) {
    if ( mor1.dom() != f1.dom() || mor2.dom() != f2.dom() )
        throw new CatError
            ("BinCoproductFinSet.univ: Wrong domain");
    if ( f1.codom() != f2.codom() )
        throw new CatError
            ("BinCoproductFinSet.univ: Wrong codomain");
    and we define

    $u(c) = \begin{cases} f_1(x) & \text{if } c \in m_1[O_1] \land c = m_1(x) \\
                          f_2(x) & \text{if } c \in m_2[O_2] \land c = m_2(x) \end{cases}$

    FinSetMap u = c.makeMapImpl();
    Iterator iter = s1.iterator();
    Object e;
    FinSetMap f1map = ((MorFinSet) f1).fct;
    while ( iter.hasNext() ) {
```
3.4 Coequalizer

Finally, we consider the coequalizer. Again, we provide the implementation with some local fields to improve the efficiency:

```java
public class CoequalizerFinSet extends Coequalizer {
    private CatFinSet c;
    private FinSetMap fct1, fct2, fct;
    private FinSet set1, set2, set;

    Some of these fields and some of the fields in the superclass can be set by the constructor at the very beginning:

    public CoequalizerFinSet (MorFinSet mor1, MorFinSet mor2) {
        this.cat = mor1.isIn();
        c = (CatFinSet) cat;
        this.mor1 = mor1;
        this.mor2 = mor2;
        obj1 = mor1.dom();
        set1 = ((ObjFinSet) obj1).asSet();
        obj2 = (ObjFinSet) mor1.codom();
        set2 = ((ObjFinSet) obj2).asSet();
        fct1 = mor1.fct;
        fct2 = mor2.fct;
    }

    If the morphisms mor1 and mor2 are parallel, we can construct the coequalizer:\footnote{We can skip this test if we use a pair of morphisms as the parameter and add the test in that class.}

    if ( !obj1.equals(mor2.dom())
        | !obj2.equals(mor2.codom()) ) {
        throw new CatError
            ("CoequalizerFinSet: Morphisms not parallel");
    }
}
```

The implementation of the coequalizer in $\mathbf{Set}$ is a little bit sophisticated:

**Lemma 10 (Coequalizer in $\mathbf{Set}$):** In $\mathbf{Set}$, we can construct the coequalizer of two morphisms $m_1, m_2 : O_1 \to O_2$ as a system of equivalence classes:

\[ O_1 \to O_2 \]
(a) Consider the relation \( R := \{(m_1(x), m_2(x)) \mid x \in O_1\} \).

(b) Construct the finest equivalence relation \( \bar{R} \) containing \( R \), i.e.,

\[
(y, z) \in R \Rightarrow (y, z) \in \bar{R} \\
(y, z) \in R \land (z, w) \in R \Rightarrow (y, w) \in \bar{R} \\
(y, z) \in \bar{R} \Rightarrow (z, y) \in \bar{R}
\]

(c) Define \( O := \{[y] \mid y \in O_2\} \) and \( m(y) := [y] \) where \([y]\) denotes the class of all elements equivalent to \( y \).

The main point is constructing the equivalence classes. We start with a set of
classes each of which contains exactly one element \( \text{set2} \):

```java
set = c.makeSetImpl();
Iterator iter = set2.iterator();
while ( iter.hasNext() ) {
    FinSet m = c.makeSetImpl();
    m.add(iter.next());
    set.add(m);
}
```

Then, we run through \( \text{set1} \) looking for the classes containing \( m_1(e) \) and \( m_2(e) \),
respectively, and we put them together by the method \( \text{find_unite} \):

```java
iter = set1.iterator();
Object e, e1, e2;
while ( iter.hasNext() ) {
    e = iter.next();
    e1 = fct1.getValue(e);
    e2 = fct2.getValue(e);
    set = set.find_unite(e1, e2);
}
```

The coequalizer maps each element \( e \) of \( s_2 \) onto the equivalence class which
contains it:

```java
fct = c.makeMapImpl();
iter = set2.iterator();
while ( iter.hasNext() ) {
    e = iter.next();
    fct.add(e, set.find(e));
}
```

The set we have constructed is a set of sets, since each element is an equivalence
class that for its part, is implemented as a set. This does not lead to any problems
since our implementation of sets allows them to contain any Java object.\(^{12}\)

The final step is as usual:

```java
obj = new ObjFinSet(c, set);
mor = new MorFinSet(c,
```

\(^{12}\) Alternatively, we can represent each equivalence class by a distinguished element.
We have used this solution in our Haskell version [4].
If we have a $q' : B \rightarrow C'$ with $q' \cdot m_1 = q' \cdot m_2$, the universal morphism $u$ is defined by $u([y]) := q'(y)$. We implement this definition step by step by adding $m(y) \mapsto q'(y)$ to the map.

```java
public Mor univ (Mor q ) {
    if ( !(q.compose(mor1)).commutes(q.compose(mor2)) ) {
        throw new CatError("Coequalizer not applicable");
    }
    FinSetMap qfct = ((MorFinSet) q).fct;
    FinSetMap ufct = c.makeMapImpl();
    Iterator iter = set2.iterator();
    while ( iter.hasNext() ) {
        Object e = iter.next();
        ufct.add(fct.getValue(e), qfct.getValue(e));
    }
    return new MorFinSet(c, (ObjFinSet) obj,
                         ufct, (ObjFinSet) q.codom());
}
```

Some of the pairs added to $ufct$ may have the same left-hand component. But the proof of the lemma shows that these pairs also have the same right-hand component. The function $\text{add}$ is implemented in such a way that no duplicates occur in the result.

## 4 The Category Graph

### 4.1 Definition

The class $\text{CatFinGraph}$ and all the other classes concerning graphs are collected in the package $\text{category.catfingraph}$. Since graphs are based on sets, the constructor of the class $\text{CatFinGraph}$ must define an implementation of sets that is used to realize the set of edges and the set of nodes. Of course, we also provide this class with a pushout creator:

```java
public class CatFinGraph implements CatWithColimits {
    protected CatFinSet baseCat;
    private PushoutCreator po;

    public CatFinGraph (CatFinSet baseCat) {
        this.baseCat = baseCat;
        po = new PushoutCreator(this);
    }
```

The public methods include access to constructing pushouts as well as the construction of the three basic colimits. This part is analogous to the class $\text{CatFinSet}$.
public Pushout pushout (Mor f, Mor g) {
    return po.pushout(f,g);
}

public InitialObj makeInitialObj() {
    return new InitialObjFinGraph(this);
}

public BinCoproduct makeBinCoproduct(Obj o1, Obj o2) {
    return new BinCoproductFinGraph((ObjFinGraph) o1,
           (ObjFinGraph) o2);
}

public Coequalizer makeCoequalizer(Mor m1, Mor m2) {
    return new CoequalizerFinGraph((MorFinGraph) m1,
           (MorFinGraph) m2);
}

These methods are used only as an interface to call the detailed constructions. Of course, you can define all the details here, but this makes the class a monster and does not conform with the principle to keep classes small.

A graph is mainly given by a pair of set morphisms. In [3, Chapter 2], we have used the following definition:

**Definition 11 (Graph):** A graph is a quadruple \( G = (E, V, s, t) \) with \( E, V \in \text{Obj}_\text{Set} \) and \( s, t \in \text{Mor}_\text{Set}(E, V) \). \( E \) is called the set of edges, \( V \) the set of nodes or vertices. Function \( s \) assigns a source node to each edge and function \( t \) a target node.

In the next subsection, we shall implement the details of this definition. Here, we restrict attention to adapting this definition to the interface \( \text{Cat} \):

```java
public ObjFinGraph makeObj(Object pair) {
    PairOfMor p = (PairOfMor) pair;
    return makeObj( (MorFinSet) p.first(),
           (MorFinSet) p.second() );
}

protected ObjFinGraph makeObj(MorFinSet s, MorFinSet t) {
    return new ObjFinGraph(this, s, t);
}
```

The second method can be used inside the package without unnecessary casts. Please note that the set of edges and the set of nodes are implicitly given as the domain and the codomain of the morphisms \( s \) and \( t \). Therefore, we can omit them in the constructor, but we have to check them in implementing the details.

Analogously, we adapt the definition of graph morphisms:
Definition 12 (Graph morphism): A graph morphism \( f : G \to H \) is a pair 
\( f = (f_E : E_G \to E_H, f_V : V_G \to V_H) \) of mappings such that 
\( f_V \cdot s_G = s_H \cdot f_E \wedge f_V \cdot t_G = t_H \cdot f_E. \)

This definition can be immediately translated into Java:

```java
public MorFinGraph makeMor(Obj from, Arrow f, Obj to) {
    PairOfMor a = (PairOfMor) f;
    return new MorFinGraph(this, (ObjFinGraph) from,
                           (MorFinSet) a.first(),
                           (MorFinSet) a.second(),
                           (ObjFinGraph) to);
}

protected MorFinGraph makeMor(ObjFinGraph from,
                              MorFinSet fe, MorFinSet fv, ObjFinGraph to) {
    return new MorFinGraph(this, from, fe, fv, to);
}
```

Defining graph objects as well as graph morphisms, we use pairs of set morphisms. We implement a more general class, and put it into the package `category`:

```java
public class PairOfMor implements Arrow {
    protected Cat cat;
    protected Mor fst, snd;

    public Cat isIn() {
        return cat;
    }

    public PairOfMor(Mor fst, Mor snd) {
        this.cat = fst.isIn();
        this.fst = fst;
        this.snd = snd;
    }

    protected Cat cat;
    protected Mor fst, snd;

    public Mor first() { return fst; }
    public Mor second() { return snd; }

    and a suitable test for equality:
    public boolean equals(Arrow f) {
        PairOfMor pair = (PairOfMor) f;
        return fst.commutes(pair.first())
               && snd.commutes(pair.second());
    }

    The reader may ask why we do not require the domains and codomains of the components to be equal. We use pairs of morphisms also to implement graph morphisms; in that case, the objects are not equal.

\( E_G \) denotes the edges of graph \( G \), \( E_H \) the edges of \( H \), \( E' \) the edges of \( G' \), \( E_1 \) the edges of \( G_1 \), etc.
4.2 Graph Objects and Graph Morphisms

In the class \texttt{CatFinGraph}, we have implemented only the interface to define graph objects. The details of Definition 11 are given in the class \texttt{ObjFinGraph}. It immediately reflects the definition:

\begin{verbatim}
public class ObjFinGraph extends Obj {

    private ObjFinSet edges;
    private ObjFinSet nodes;
    private MorFinSet sourceMap;
    private MorFinSet targetMap;

    The constructor extracts these data from the given morphisms and checks the details:

    public ObjFinGraph(Cat cat, MorFinSet s, MorFinSet t) {
        this.cat = cat;
        this.edges = (ObjFinSet) s.dom();
        this.nodes = (ObjFinSet) s.codom();
        this.sourceMap = s;
        this.targetMap = t;
        if ( !edges.equals(t.dom()) ) { throw new CatError
            ("ObjFinGraph: Domains don’t match");
        }
        if ( !nodes.equals(t.codom()) ) { throw new CatError
            ("ObjFinGraph: Codomains don’t match");
        }
    }

    Special methods allow accessing the components of a graph as mentioned in the definition:

    public ObjFinSet edgesOfGraph() { return edges; }
    public ObjFinSet nodesOfGraph() { return nodes; }
    public MorFinSet srcOfGraph() { return sourceMap; }
    public MorFinSet tgtOfGraph() { return targetMap; }

    Equality test leads to a problem. From a theoretical point of view, we need a test for isomorphism. As you know, this is a very expensive task, and therefore, we simply test equality of the components:

    protected boolean equalsSemant(Obj g) {
        ObjFinGraph gg = (ObjFinGraph) g;
        return this.sourceMap.commutes(gg.sourceMap)
            && this.targetMap.commutes(gg.targetMap);
    }

    Finally, we have to implement the identity morphism associated with each graph. It is simply given by the identity morphisms of the sets of edges and nodes:

    public Mor computeId() {
        return ((CatFinGraph)cat).makeMor
            (this, (MorFinSet)edges.id(),
            (MorFinSet)nodes.id(), this);
    }
\end{verbatim}
Now, we switch over to graph morphisms. The class definition includes the graphs $G$ and $H$, the set morphism $f_E$ between the edges, and the set morphism $f_V$ between the nodes:

```
public class MorFinGraph extends Mor {

    private MorFinSet edgeMor;
    private MorFinSet nodeMor;
```

![Diagram]

The conditions of Definition 12 can be illustrated graphically. The constructor has to check the commutativity of these diagrams:

```
public MorFinGraph(Cat cat, ObjFinGraph dom,
    MorFinSet fe, MorFinSet fv, ObjFinGraph codom) {

    this.cat = cat;
    this.dom = dom;
    this.codom = codom;
    edgeMor = fe;
    nodeMor = fv;
    this.arrow = new PairOfMor(edgeMor, nodeMor);

    if ( !codom.srcOfGraph().compose(fe).
        commutes(fv.compose(dom.srcOfGraph()))
    ) {
        throw new CatError
            ("MorFinGraph: src-diagram not commutative");
    }

    if ( !codom.tgtOfGraph().compose(fe).
        commutes(fv.compose(dom.tgtOfGraph()))
    ) {
        throw new CatError
            ("MorFinGraph: tgt-diagram not commutative");
    }
}
```

For reason of efficiency, we allowed immediately accessing $f_E$ and $f_V$, but we restrict this to the methods belonging to the graph package:

```
protected MorFinSet getEdgeMor() { return edgeMor; }
protected MorFinSet getNodeMor() { return nodeMor; }
```

Finally, we need an implementation of the method `composeWithoutTest`, that we have introduced in the abstract class `Mor`:

```
protected Mor composeWithoutTest(Mor m) {
    MorFinGraph mor2 = (MorFinGraph) m;
```
MorFinSet fe = (MorFinSet) edgeMor.compose(mor2.edgeMor);
MorFinSet fv = (MorFinSet) nodeMor.compose(mor2.nodeMor);
return ((CatFinGraph)cat).makeMor
((ObjFinGraph)m.dom(), fe, fv, this.to);
}

Please note that we don’t need a special implementation of commutes due to existence of the equals-method in class Arrow.

4.3 Colimits in Graph

The class CatFinGraph has already adapted the methods to construct the three basic colimits to the generic definitions by forwarding them to methods characteristic of the case of graphs. All the colimit constructions in the category of graphs are based on constructing the colimits for edges and nodes in Set, separately. Then, the constructed objects must be made graphs by defining the source and the target functions in a suitable way.

First, we consider the coproduct:

```java
public class BinCoproductFinGraph extends BinCoproduct {

    private CatFinGraph c;
    private CatFinSet basecat;
    private BinCoproductFinSet coprodEdges, coprodNodes;

    The constructor defines the coproducts for edges and nodes, separately:
    public BinCoproductFinGraph (ObjFinGraph o1, ObjFinGraph o2) {
        this.cat = o1.isIn();
        c = (CatFinGraph) cat; /* avoids casting */
        this.basecat = c.baseCat;
        /* Coproduct for nodes */
        coprodNodes = (BinCoproductFinSet) basecat.
        makeBinCoproduct(o1.nodesOfGraph(), o2.nodesOfGraph());
        ObjFinSet cpnodes = (ObjFinSet) coprodNodes.obj();
        MorFinSet m1nodes = (MorFinSet) coprodNodes.mor1();
        MorFinSet m2nodes = (MorFinSet) coprodNodes.mor2();
        /* Coproduct for edges */
        coprodEdges = (BinCoproductFinSet) basecat.
        makeBinCoproduct(o1.edgesOfGraph(), o2.edgesOfGraph());
        ObjFinSet cpedges = (ObjFinSet) coprodEdges.obj();
        MorFinSet m1edges = (MorFinSet) coprodEdges.mor1();
        MorFinSet m2edges = (MorFinSet) coprodEdges.mor2();
    }

    Then, the coproduct property for the edges can be used to define the source and
    the target functions of the coproduct object in Graph:
```
Implementing the universal property, we have to ensure that the given morphisms satisfy the condition of Definition 2:

```java
public Mor univ(Mor f1, Mor f2) {
    MorFinGraph f1g = (MorFinGraph) f1;
    MorFinGraph f2g = (MorFinGraph) f2;
    if (mor1.dom() != f1.dom() || mor2.dom() != f2.dom())
        throw new CatError("Morphisms not compatible");
    if (f1.codom() != f2.codom())
        throw new CatError("Morphisms not compatible");

    MorFinSet cpsource = (MorFinSet) coprodEdges.univ(
        m1nodes.compose(o1.srcOfGraph()),
        m2nodes.compose(o2.srcOfGraph()));
    MorFinSet cptarget = (MorFinSet) coprodEdges.univ(
        m1nodes.compose(o1.tgtOfGraph()),
        m2nodes.compose(o2.tgtOfGraph()));
    cp = c.makeObj(cpsource, cptarget);
    mor1 = c.makeMor(o1, new PairOfMor(m1edges, m1nodes), cp);
    mor2 = c.makeMor(o2, new PairOfMor(m2edges, m2nodes), cp);
    return c.makeMor(cp, new PairOfMor(
        coprodEdges.univ(f1g.getEdgeMor(),
        f2g.getEdgeMor()),
        coprodNodes.univ(f1g.getNodeMor(),
        f2g.getNodeMor())),
        f1.codom());
}
```

Here, we have no problems with source function or target function, since we do not construct any new objects.

Now, we consider the coequalizer. Again, we can proceed component by component. In this case, however, we have to make these results a graph by constructing a source and a target function.

```java
public class CoequalizerFinGraph extends Coequalizer {
    private CatFinGraph c;
    private CatFinSet basecat;

    private CoequalizerFinSet edgecoeq, nodecoeq;
    private ObjFinGraph o1g, o2g;
```
As usual, the constructor first copies the objects and morphisms involved in the construction and it checks that the morphisms are parallel.\footnote{Another way is to use a pair of morphisms as the parameter.}

\begin{verbatim}
public CoequalizerFinGraph ( MorFinGraph m1, MorFinGraph m2 ) {
    this.cat = m1.isIn();
    c = (CatFinGraph) cat;
    this.basecat = c.baseCat;
    mor1 = m1;
    mor2 = m2;
    obj1 = m1.dom(); obj2 = m1.codom();
    o1g = (ObjFinGraph) obj1; o2g = (ObjFinGraph) obj2;
    if ( !obj1.equals(mor2.dom())
        | !obj2.equals(mor2.codom()) ) {
        throw new CatError("Morphisms not parallel");
    }

    Separately considering edges and nodes yields the edges and the nodes of the coequalizer object as well as the mappings from the second object to the result:
    
    /* Coequalizer for edges */
    MorFinSet me1 = m1.getEdgeMor();
    MorFinSet me2 = m2.getEdgeMor();
    edgecoeq = (CoequalizerFinSet)
        basecat.makeCoequalizer(me1, me2);
    MorFinSet me = (MorFinSet) edgecoeq.mor();
    ObjFinSet e = (ObjFinSet) edgecoeq.obj();
    /* Coequalizer for nodes */
    MorFinSet mv1 = m1.getNodeMor();
    MorFinSet mv2 = m2.getNodeMor();
    nodecoeq = (CoequalizerFinSet)
        basecat.makeCoequalizer(mv1, mv2);
    MorFinSet mv = (MorFinSet) nodecoeq.mor();
    ObjFinSet v = (ObjFinSet) nodecoeq.obj();

    The separately constructed sets of edges and of nodes must be made a graph by defining the source and the target functions using the universal property of the coequalizer of the edge morphisms:

    MorFinSet s =
        (MorFinSet) edgecoeq.univ(mv.compose(o2g.srcOfGraph()));
    MorFinSet t =
        (MorFinSet) edgecoeq.univ(mv.compose(o2g.tgtOfGraph()));
    obj = c.makeObj(s, t);
    /* Making (qe, qv) a graph morphism */
\end{verbatim}
Finally, we need the universal property:

\[
\begin{array}{c}
E_1 \rightarrow E_2 \\
V_1 \rightarrow V_2
\end{array}
\]

\[
\begin{array}{c}
E \\
V
\end{array}
\]

\[
\begin{array}{c}
E' \\
V'
\end{array}
\]

\[
\begin{array}{c}
s_2 \\
q_E \\
q_V \\
\alpha_E \\
\alpha_V
\end{array}
\]

\[
\begin{array}{c}
v_2 \\
v_1 \\
v
\end{array}
\]

\[
\begin{array}{c}
E' \\
v
\end{array}
\]

\[
\begin{array}{c}
E \\
v
\end{array}
\]

\[
\begin{array}{c}
V \\
v
\end{array}
\]

\[
\begin{array}{c}
V' \\
v
\end{array}
\]

\[
\begin{array}{c}
q_E \\
q_V
\end{array}
\]

\[
\begin{array}{c}
s_2 \\
\alpha_E \\
\alpha_V
\end{array}
\]

\[
\begin{array}{c}
v_2 \\
v_1 \\
v
\end{array}
\]

\[
\begin{array}{c}
E' \\
v
\end{array}
\]

\[
\begin{array}{c}
E \\
v
\end{array}
\]

\[
\begin{array}{c}
V \\
v
\end{array}
\]

\[
\begin{array}{c}
V' \\
v
\end{array}
\]

\[
\begin{array}{c}
q_E \\
q_V
\end{array}
\]

Since the graphs involved in the universal construction are given, it is not necessary to consider source and target functions. Furthermore, \texttt{makeMor} ensures the commutativity condition of Definition 12, although this check is not necessary because it is a consequence of the construction.

The initial object in \texttt{Graph} is given by the empty set of edges and the empty set of nodes, i.e., the initial object in \texttt{Set} together with empty source and target functions:

\[
\text{public class InitialObjFinGraph extends InitialObj} \{
\]

\[
\text{private CatFinGraph c;}
\]

\[
\text{private CatFinSet basecat;}
\]

\[
\text{public InitialObjFinGraph (CatFinGraph cat) \{}
\]

\[
\text{this.cat = cat;}
\]

\[
\text{c = cat;}
\]

\[
\text{this.basecat = cat.baseCat;}
\]

\[
\text{InitialObjFinSet emptyObj =}
\]

\[
\text{(InitialObjFinSet) basecat.makeInitialObj();}
\]

\[
\text{MorFinSet emptyArrow =}
\]

\[
\text{(MorFinSet) emptyObj.univ(emptyObj.obj());}
\]

\[
\text{init = cat.makeObj(emptyArrow, emptyArrow);}
\]

\[
\text{\}}
\]

The universal property is just as simple:
This completes the implementation of the category of graphs. In [3, Chapter2], we have given an alternative construction of the pushout diagram using the pushout diagrams for edges and nodes. You can also implement this construction here and override the generic method.

5 The Category $\textit{Set}^{\textit{inc}}$

5.1 Definition

We now consider another category that is of theoretical interest, since it has some unusual properties. As in $\textit{Set}$, the objects of $\textit{Set}^{\textit{inc}}$ are the sets. But now, we define the morphisms in another way: There is exactly one morphism from $A$ to $B$ if $A$ is a subset of $B$. Otherwise, $\text{Mor}_{\textit{Set}^{\textit{inc}}}(A, B)$ is empty. Existence of composition follows from $A \subseteq B \land B \subseteq C \Rightarrow A \subseteq C$ and existence of identities from $A \subseteq A$.

The definition of the class $\text{CatSetIncl}$ is analogous with $\text{CatFinSet}$:

```java
public class CatSetIncl implements CatWithColimits {
    public ObjSetIncl makeObj(Object o) {
        return new ObjSetIncl(this, (FinSet) o);
    }

    public MorSetIncl makeMor(Obj from, Arrow f, Obj to) {
        return new MorSetIncl(this, (ObjSetIncl) from, (ObjSetIncl) to);
    }
}
```

We can not use the same objects as in $\text{CatFinSet}$, since the categorical objects are not simply sets, e.g., they include the identity morphism, which is of course, different from the identity morphism in $\textit{Set}$.

Since $\textit{Set}^{\textit{inc}}$ has colimits, the class also must implement the construction of the basic colimits and must link these constructors with the methods defined in the interface $\text{CatWithColimits}$:

---

15 On the other hand, $\textit{Set}^{\textit{inc}}$ does not have all limits, since there is no terminal object as long as we consider only finite sets.
public InitialObj makeInitialObj() {
    return new InitialObjSetIncl(this);
}

public BinCoproduct makeBinCoproduct(Obj o1, Obj o2) {
    return new BinCoproductSetIncl((ObjSetIncl) o1, (ObjSetIncl) o2);
}

public Coequalizer makeCoequalizer(Mor m1, Mor m2) {
    return new CoequalizerSetIncl((MorSetIncl) m1, (MorSetIncl) m2);
}

Then, the other colimits, such as the pushout, can be constructed in a generic way. For this, the constructor of class CatSetIncl must provide the class with a pushout creator:

    private PushoutCreator po;

    public CatSetIncl () {
        po = new PushoutCreator(this);
    }

    public Pushout pushout (Mor f, Mor g) {
        return po.pushout(f,g);
    }

5.2 Objects and Morphisms in Setincl

Contrary to the previous examples, the arrows between two objects are rather trivial: An arrow exists or does not exist at all. Therefore, we need only one object in the class ArrowSetIncl:

    public class ArrowSetIncl implements Arrow {
        public static ArrowSetIncl arrow = null;
        public boolean equals (Arrow f) { return true; }
    }

The constructor MorSetIncl must test that the set underlying the first object is a subset of the second:

    public class MorSetIncl extends Mor {

        public MorSetIncl(CatSetIncl cat, ObjSetIncl dom, ObjSetIncl codom) {
            this.cat = cat;
            this.dom = dom;
            this.codom = codom;
            this.arrow = ArrowSetIncl.arrow;
        }
    }
if (!dom.asSet().subsetOf(codom.asSet())) {
    throw new CatError("MorSetIncl: No morphism");
}
}

Composing morphisms in \(\text{Setincl}\) simply means removing the object in the middle:\(^\text{16}\)

```java
protected Mor composeWithoutTest(Mor m) {
    return ((CatSetIncl)cat).makeMor
        (m.dom(), ArrowSetIncl.arrow, this.codom);
}
```

The constructor `ObjSetIncl` transforms a finite set into a categorical object, and `asSet` is the inverse method:

```java
public class ObjSetIncl extends Obj {
    private FinSet set;
    public ObjSetIncl(CatSetIncl cat, FinSet set) {
        this.cat = cat;
        this.set = set;
    }
    public FinSet asSet() { return set; }
}
```

Identity in \(\text{Setincl}\) is given by the fact that each set is a subset of itself:

```java
protected Mor computeId() {
    return ((CatSetIncl)cat).makeMor
        (this, ArrowSetIncl.arrow, this);
}
```

The semantical equality is the equality of the underlying sets:

```java
public boolean equalsSemant(Obj s) {
    return set.equals(((ObjSetIncl) s).asSet());
}
```

### 5.3 Colimits in \(\text{Setincl}\)

The fact that morphisms coincide with inclusion allows a very simple characterization of the colimits in \(\text{Setincl}\). The initial object is the set that is contained in each set, i.e., the empty set:

```java
public class InitialObjSetIncl extends InitialObj {
    private FinSet initImpl;
    public InitialObjSetIncl (CatSetIncl cat) {
        this.cat = cat;
    }
}
```

\(^{16}\) You cannot omit the parentheses in \(\text{m.dom()}\) [5, Sect. 6.6.7]
The coproduct object \( C \) is the object that is a superset of both \( A \) and \( B \) and is contained in all sets with this property, i.e., \( C \) is the union \( C = A \cup B \):

```java
public class BinCoproductSetIncl extends BinCoproduct {
    private CatSetIncl c;
    private FinSet s1, s2, cpset;

    public BinCoproductSetIncl (ObjSetIncl o1, ObjSetIncl o2) {
        this.cat = o1.isIn();
        c = (CatSetIncl) cat;
        s1 = o1.asSet();
        s2 = o2.asSet();
        cpset = s1.union(s2);
        cp = new ObjSetIncl(c, cpset);
        mor1 = new MorSetIncl(c, o1, (ObjSetIncl) cp);
        mor2 = new MorSetIncl(c, o2, (ObjSetIncl) cp);
    }

    public Mor univ ( Mor f1, Mor f2 ) {
        if ( mor1.dom() != f1.dom() || mor2.dom() != f2.dom() )
            throw new CatError
                ("BinCoproductSetIncl.univ: Wrong domain");
        if ( f1.codom() != f2.codom() )
            throw new CatError
                ("BinCoproductSetIncl.univ: Wrong codomain");
        return new MorSetIncl(c, (ObjSetIncl) cp, (ObjSetIncl) f1.codom());
    }
}
```

The coequalizer is the identity since parallel morphisms are identical:

```java
public class CoequalizerSetIncl extends Coequalizer {
    private CatSetIncl c;
    private FinSet set1, set2, set;

    public CoequalizerSetIncl (MorSetIncl m1, MorSetIncl m2) {
```
this.cat = m1.isIn();
c = (CatSetIncl) cat;
mor1 = m1;
mor2 = m2;
obj1 = m1.dom();
obj2 = m1.codom();
obj = obj2;
mor = obj2.id();
}

Testing \( q \cdot m_1 = q \cdot m_2 \) can be reduced to check that \( q \) starts at \( o_2 \):

public Mor univ (Mor q ) {
   if ( !(q.dom().equals(obj2)) ) {
      throw new CatError("Coequalizer not applicable");
   }
   return new MorSetIncl(c,
      (ObjSetIncl) obj, (ObjSetIncl) q.codom());
}

If we want, we can override the standard pushout construction by a special solution. Since the coequalizer is the identity, the pushout construction is essentially given by the coproduct construction.

6 Extensions

We have already mentioned (p. 4) that we can extend the concept by considering special properties of morphisms such as monomorphisms and epimorphisms:

\[
\begin{align*}
\text{public abstract boolean isMonomorphic();} \\
\text{public abstract boolean isCoretraction();} \\
\text{public abstract boolean isEpimorphic();} \\
\text{public abstract boolean isRetraction();}
\end{align*}
\]

We add these definitions to the class \( \text{Mor} \). They are sufficient to identify isomorphisms on this general level:

\[
\begin{align*}
\text{public boolean isIsomorphic() } & \quad \text{\{ return (isMonomorphic() && isRetraction()) } \\
   & \quad \text{|| (isEpimorphic() && isCoretraction()) } \}
\end{align*}
\]

At this point, we can avoid multiple inheritance.

Of course, we can check these properties only when implementing special categories, e.g., in \( \text{MorFinSet} \):

\[
\begin{align*}
\text{public boolean isMonomorphic() } & \quad \text{\{ return fct.injective(); } \}
\text{public boolean isCoretract() } \quad \text{\{ return fct.injective() } \\
   & \quad \text{&& ( ((ObjFinSet) dom).asSet().size() != 0) } \\
   & \quad \text{|| ( ((ObjFinSet) codom).asSet().size() == 0 ));}
\end{align*}
\]
(The only monomorphism in \( \mathit{Set} \) that is not a coretraction is the function mapping the empty set into a nonempty one.)

The construction of derivation steps (see [3, Chapter 4]) requires constructing pushout complements. For this, we can add a method

\[
\text{poCompl}(\text{Mor } p, \text{ Mor } g) \{ \ldots \}
\]

to the class \texttt{PushoutCreator}. In the case of \( \mathcal{E} \)-\( \mathcal{M} \)-factorizable categories we can subdivide the task into smaller subtasks that are easier to comprehend and to implement:

\[
\begin{align*}
\text{poCompl}(\text{MonoMor } p, \text{ EpiMor } g) & \{ \ldots \} \\
\text{poCompl}(\text{MonoMor } p, \text{ MonoMor } g) & \{ \ldots \} \\
\text{poCompl}(\text{EpiMor } p, \text{ EpiMor } g) & \{ \ldots \} \\
\text{poCompl}(\text{EpiMor } p, \text{ Monomor } g) & \{ \ldots \}
\end{align*}
\]

For this, we need special subclasses or interfaces. Of course, it is possible to define \texttt{MonoMor} and \texttt{EpiMor} as subclasses of \texttt{Mor} in the package \texttt{category}. But, we get a problem in implementing special categories, e.g., \texttt{MonoMorFinSet} becomes a subclass of both \texttt{MonoMor} and \texttt{MorFinSet}. This means that we have to use interfaces:

\[
\begin{align*}
\text{public interface MonoMor } & \{ \} \\
\text{public interface EpiMor } & \{ \}
\end{align*}
\]

Let us consider the monomorphisms in \( \mathit{Set} \) as an example:

\[
\text{public class MonoMorFinSet extends MorFinSet implements MonoMor } \{ \\
\text{public MonoMorFinSet(CatFinSet cat, ObjFinSet from, FinSetMap fct, ObjFinSet to) } \{ \\
\text{super(cat, from, fct, to);} \\
\text{if ( !fct.injective() ) } \{ \\
\text{throw new CatError("MonoMorFinSet: Mapping not injective");} \\
\}
\}
\]

Trivially, we can override the method \texttt{isMonomorphic} in this case, since the explicit test is not necessary:

\[
\text{public boolean isMonomorphic()} \{ \text{return true; } \}
\]

The analogous definitions for epimorphisms are left to the reader.

7 Conclusion

A problem we have in implementing the categorical concepts in Java is that Java forbids multiple inheritance. The systematic organization of the categorical concepts, however, leads to such a hierarchy in a natural way. We have considered two aspects.

We have explained in detail the constructions of colimits; for reason of space, we have omitted the limit constructions. The categories of interest have colimits
as well as limits. Therefore, we have defined the categories with colimits and the categories with limits as interfaces, and we have implemented the categorical constructions in separate factory classes, which we must explicitly import into each category of interest. This is a little bit expensive; an advantage of this structure, however, is that we can keep small the class definitions.

Another concept that leads to multiple inheritance is the hierarchy of morphisms: An isomorphism is both a monomorphism and an epimorphism. In this case, we have chosen another way: We have defined monomorphisms, epimorphisms, retractsions, and coretractsions as classes to allow a convenient formulation of complement constructions in $E-M$-factorizable categories. This means that we have to abandon the implementation of the class of isomorphisms. Due to a theoretical result, however, we can implement the isomorphism test immediately in the class of morphisms.

Another problem is that we have to use cast operations in implementing specific categories, as you have seen in the sections describing $Set$ and $Graph$. We can avoid them by using parameterized classes and methods, i.e., the categories are parameterized by the types of objects and morphisms. Let us consider $CatFinSet$ that we have described on page 12. The parameterized version can be read as follows:

```java
public class FinSetCat
    implements CatWithColimits<FinSetObj, FinSetMor> {

    private PushoutCreator<FinSetObj, FinSetMor, FinSetCat> po;

    public FinSetCat() {
        po = new PushoutCreator<FinSetObj, FinSetMor, FinSetCat>(this);
    }

    public Pushout<FinSetObj, FinSetMor> pushout(FinSetMor f, FinSetMor g) {
        return po.pushout(f, g);
    }

    public FinSetMor makeMor(FinSetObj from, Applier app, FinSetObj to) {
        return new FinSetMor(this, from, app, to);
    }

    // (Applier is an interface an implementation of which is mapping one set to another.)

    // A great advantage of Java is the possibility to re-use identifiers of methods and constructors as long as the parameters unambiguously determine the concrete

    17 The author thanks M. Minas for implementing this alternative.
method and constructor, respectively. We have applied this technique in implementing the pushout complement as mentioned in the last section. We have also applied it to the construction of pushouts: The usual construction needs two morphisms as parameters, whereas the method constructing a new pushout by composing two pushouts has the subdiagrams as parameters. (Although the present implementation uses this trick, we have to ask for whether this second example is good programming practice.)

Comparing the Java version with the Haskell version [4], we miss the literate programming facility. In Haskell, we can combine LaTeX text and Haskell code in one file that can be processed both by the LaTeX processor and the Haskell interpreter. This means that we have only one source file: The program and its description are consistent in a natural way.

References

3. H.J. Schneider: *Graph Transformations – An Introduction to the Categorical Approach*, Preliminary version: 
   [http://www2.informatik.uni-erlangen.de/~schneide/gtbook/index.xml](http://www2.informatik.uni-erlangen.de/~schneide/gtbook/index.xml)
   (Link checked on May 1st, 2009)
4. H.J. Schneider: *Implementing the categorical approach to graph transformations with Haskell*
   [http://www2.informatik.uni-erlangen.de/Personen/schneide/gtbook/appendix-a.pdf](http://www2.informatik.uni-erlangen.de/Personen/schneide/gtbook/appendix-a.pdf)
   (Link checked on May 1st, 2009)
   [http://java.sun.com/docs/books/jls](http://java.sun.com/docs/books/jls)
   (Link checked on July 17th, 2009)
   [http://java.sun.com/javase/6/docs/api/](http://java.sun.com/javase/6/docs/api/)
   (Link checked on May 1st, 2009)