Implementing the Categorical Approach to Graph Transformations With Haskell

Hans J. Schneider
Universität Erlangen-Nürnberg – Institut für 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 Haskell, 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 [4] 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 Haskell support generic concepts, it is a promising idea to present the categorical approach to graph transformations in Haskell [1, 3]. 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. 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 [2]. Their approach is mainly based on polymorphism, whereas we can additionally take advantage of Haskell’s class concept.

The paper is organized as follows. Section 2 summarizes the general definitions of a category and of colimits, and it presents the implementation of these definitions in Haskell. By way of example, we show that categorical constructions can be easily implemented on this very abstract level. We consider the canonical construction of pushouts from coproducts and coequalizers, and the composition of pushouts. The next section describes a small interface module necessary to avoid a recursive module structure. Section 3 is concerned with the category of sets. Here, we must give concrete implementations of the objects and morphisms constituting the category and of the methods that are not defined by default. We do the same for the category of graphs in Section 4, but this is much easier.
At some few places, the order of the cited Haskell code does not agree with the optimal order to explain it systematically. The reason for this is that the LaTeX source of the present document is valid Haskell code using the \texttt{.hs}-suffix. Haskell allows us to arrange the definitions rather freely, but there are some few constraints, e.g., the export clause appears in the module head, although it makes sense to explain the export after it is defined.

2 The Generic Modules

We start with summarizing some basic definitions and constructions of category theory and translate them into Haskell straightforwardly. The presentation is based on our textbook [4, Chapter 2], where the reader can find more detailed explanations and examples.

2.1 Category

The module \texttt{Category} reflects 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:

\textbf{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:\footnote{In formulae, we usually omit the index \( C \), since the category under consideration is obvious.}

1. \( \text{dom}_C, \text{codom}_C : \text{Mor}_C \rightarrow \text{Obj}_C \)
2. \( \cdot_C : \text{Mor}_C \times \text{Mor}_C \rightarrow \text{Mor}_C \) is a partial function with the following properties:
   - \( g \cdot_C f \) exists if and only if \( \text{dom}_C(f) = \text{dom}_C(g) \).
   - \( \text{dom}_C(g \cdot_C f) = \text{dom}_C(f) \land \text{codom}_C(g \cdot_C f) = \text{codom}_C(g) \).
   - 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 \((\forall A, B \in \text{Obj}_C)(\forall f : A \rightarrow B)(\text{id}_B \cdot_C f = f = f \cdot_C \text{id}_A)\).

Before we can formulate this in Haskell, we need objects and morphisms. The only assumption objects and morphisms must satisfy is that there is a test for (in-)equality:

\begin{verbatim}
module Category where

class (Eq o) => Obj o

\end{verbatim}
class (Eq m) => Mor m where
  isMonomorphic :: m -> Bool
  isEpimorphic :: m -> Bool

The methods \texttt{isMonomorphic} and \texttt{isEpimorphic} are not used by the present version of the program, but they are provided for later use.

Each morphism has a domain object and a codomain object:

\begin{verbatim}
class Category c where
dom:: (Obj o, Mor m) => c o m -> m -> o
codom:: (Obj o, Mor m) => c o m -> m -> o
\end{verbatim}

Conversely, there is an identity morphism associated with each object:

\begin{verbatim}
ident:: (Obj o, Mor m) => c o m -> o -> m
\end{verbatim}

Remark: Please note that the first parameter of each method specifies the category in which the operation is to be performed. The type \texttt{c o m} of categories depends on the type \texttt{o} of the objects and of the type \texttt{m} of the morphisms.

The main feature of the definition is that morphisms may be composed if they fit together:

\begin{verbatim}
compose:: (Obj o, Mor m) => c o m -> m -> m -> m
composable:: (Obj o, Mor m) => c o m -> m -> m -> Bool
\end{verbatim}

In most categorical proofs composable immediately follows from the constructions. In these cases, it is not necessary to test it. Therefore, we add another method to this interface, for reason of efficiency:

\begin{verbatim}
composeWithoutTest :: (Obj o, Mor m) => c o m -> m -> m -> m
\end{verbatim}

Some of these methods can be implemented by default, i.e., without knowing the category under consideration:

\begin{verbatim}
composable c m2 m1 = (dom c m2) == (codom c m1)
composable c m2 m1 = if composable c m2 m1 then
  composeWithoutTest c m2 m1
  else error ("Category.compose: " ++
    "Morphisms don't match")
\end{verbatim}

The remaining methods must be implemented for each category separately.
2.2 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. We collect these definitions in a module Colimits:

```haskell
module Colimits where

import Category

First, we need the datatypes implementing the colimit constructions.

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

\[
(\forall O')(\forall f'_1 : O_1 \rightarrow O')(\forall f'_2 : O_2 \rightarrow O')(\exists ! u : O \rightarrow O')(f'_1 = u \cdot m_1 \land f'_2 = u \cdot m_2)
\]

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

The condition can be interpreted graphically:

```
\[
\begin{array}{c}
O_1 \\
\downarrow^{m_1}
\end{array}
\end{array} =
\begin{array}{c}
O \\
\downarrow^{m_2}
\end{array} =
\begin{array}{c}
O_2 \\
\downarrow^{f'_1}
\end{array}
\]
```

This definition can be immediately translated into a Haskell definition:\(^2\)

```haskell
data (Obj o, Mor m) =>
  Coproduct o m = Coproduct {cpObj1:: o,
                            cpObj2:: o,
                            cpObj:: o,
                            cpMor1:: m,
                            cpMor2:: m,
                            cpUniv:: m -> m -> m }
```

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

\(^2\)Haskell syntax allows us to denote a data type and a constructor by the same identifier.
Definition 3 (Coequalizer): A morphism \( m : O_2 \rightarrow O \) is called the coequalizer of a pair of parallel morphisms \( m_1, m_2 : O_1 \rightarrow O_2 \) if and only if the following conditions hold:

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

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

Again, we illustrate the factorization graphically:

\[
\begin{array}{c}
O_1 \\
\downarrow m_1 \\
O_2 \\
\downarrow m_2 \\
\downarrow m \\
O
\end{array}
\]

\[
\begin{array}{c}
O' \\
\uparrow \bar{u}
\end{array}
\]

As before, the data type definition is straight-forward:

\[
data \text{(Obj } o, \text{ Mor } m) = \rightarrow \text{ Coequalizer } o m = \text{ Coequalizer } \{\text{ceqObj1} :: o, \text{ceqObj2} :: o, \text{ceqObj} :: o, \text{ceqMor1} :: m, \text{ceqMor2} :: m, \text{ceqMor} :: m, \text{ceqUniv} :: m \rightarrow m\}
\]

Next, we consider the pushout construction starting from two morphisms with the same domain (interface object), but different codomains:

Definition 4 (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:

\[
(\forall O')(\forall \bar{g}' : O_1 \rightarrow O') \quad (\forall \bar{f}' : O_2 \rightarrow O') \quad (\bar{g}' \cdot m_1 = \bar{f}' \cdot m_2 \Rightarrow (\exists u : O \rightarrow O')(\bar{g}' = u \cdot \bar{m}_2 \land \bar{f}' = u \cdot \bar{m}_1))
\]
$O$ 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.

Here is the corresponding data type:

```haskell
data (Obj o, Mor m) =>
    Pushout o m = Pushout {poInt:: o,
                           poObj1:: o,
                           poObj2:: o,
                           poObj:: o,
                           poMor1:: m,
                           poMor2:: m,
                           poMor1b:: m,
                           poMor2b:: m,
                           poUniv:: m -> m -> m}
```

As in the previous data types, the field labels are chosen according to the diagram. In the case of the pushout data type, $\text{poMor1b}$ denotes $\overline{m}_1$, the letter $b$ indicating the bar.

Finally, we introduce the simplest colimit construction:

**Definition 5 (Initial object):** An object $O$ is called the initial object of category $\mathcal{C}$ provided that for all objects $O' \in \text{Obj}_{\mathcal{C}}$, there is exactly one morphism $O \to O'$.

```haskell
data (Obj o, Mor m) =>
    Initial o m = Initial {initObj:: o,
                           initUniv:: o -> m}
```

*Remark:* We have implemented the colimit constructions in such a way that each Haskell object represents the whole diagram including the universal property and that each component of the diagram is immediately accessible by a suitable field label. This decision makes working with the colimits more convenient than omitting all the information that can be found implicitly. (E.g., the objects could be accessed as the domain or codomain of the morphisms.)

Looking at the objects and morphisms resulting from a colimit construction, we need an appropriate show function, e.g., in the case of coproducts:

```haskell
instance (Show o, Show m, Obj o, Mor m) =>
    Show (Coproduct o m) where
    show cp = "Coproduct:
    " ++ "Morphism1: " ++ show (cpMor1 cp) ++ "\n    +++++
```

"Coproduct object: " ++ show (cpObj cp) ++
"\n" ++
"Morphism2: " ++ show (cpMor2 cp)

This is a method, which may not be adequate in each case. Implementing example categories, we can override it.

For reason of space, we have omitted the default show functions for the other colimit constructions. Nevertheless, they are included in the Haskell code.

### 2.3 Canonical Construction of Pushouts

We now introduce the subclass of categories that have colimits. A category that has initial object, coproduct, and coequalizer has all finite colimits. Therefore, we define the subclass `CatWithColimits` in such a way that the definition ensures that all instances implement these basic colimits:

```haskell
class (Category c) => CatWithColimits c where
    initial:: (Obj o, Mor m) =>
        c o m -> Initial o m
    coproduct:: (Obj o, Mor m) =>
        c o m -> o -> o -> Coproduct o m
    coequalizer:: (Obj o, Mor m) =>
        c o m -> m -> m -> Coequalizer o m

    pushout:: (Obj o, Mor m) =>
        c o m -> m -> m -> Pushout o m
```

In addition, we equip this subclass with a method constructing pushout diagrams:

```haskell
    pushout:: (Obj o, Mor m) =>
        c o m -> m -> m -> Pushout o m
```

Whereas `initial`, `coproduct`, and `coequalizer` must be implemented for each category, we can give a default implementation of `pushout` based on the following theorem:

**Theorem 6 (Canonical construction of pushouts):** A category with coproducts and coequalizers has pushouts, too.

The proof is constructive and can easily be translated into Haskell code along Figure 6a. First, we have to test that the morphisms start from the same object:

```haskell
    pushout c m1 m2
    = if (dom c m1) /= (dom c m2) then
        error "Pushout: Domains not identical"
    else Pushout {poInt = dom c m1,
```
We introduce some auxiliary objects to avoid repetitive execution of operations and to make the definition better readable:

where

\[
\begin{align*}
\text{poObj1} &= \text{obj1}, \\
\text{poObj2} &= \text{obj2}, \\
\text{poObj} &= \text{obj}, \\
\text{poMor1} &= m1, \\
\text{poMor2} &= m2, \\
\text{poMor1b} &= \text{composeWithoutTest}\ c\ r\ g', \\
\text{poMor2b} &= \text{composeWithoutTest}\ c\ r\ f', \\
\text{poUniv} &= u
\end{align*}
\]

We first construct the coproduct \((f': O_1 \rightarrow E, g': O_2 \rightarrow E, E)\) of \(O_1\) and \(O_2\):

\[
\begin{align*}
cp &= \text{coproduct}\ c\ \text{obj1}\ \text{obj2} \\
f' &= \text{cpMor1}\ cp \\
g' &= \text{cpMor2}\ cp
\end{align*}
\]

Then, we construct the coequalizer \(r: E \rightarrow O\) of \(g' = g' \cdot m_2\) and \(f'' = f' \cdot m_1\):

\[
\begin{align*}
f''' &= \text{composeWithoutTest}\ c\ f'\ m1 \\
g''' &= \text{composeWithoutTest}\ c\ g'\ m2 \\
\text{ceq} &= \text{coequalizer}\ c\ f''\ g''' \\
\text{obj} &= \text{ceqObj}\ ceq \\
r &= \text{ceqMor}\ ceq
\end{align*}
\]

Finally, we define \(\bar{m}_1 := r \cdot g'\) and \(\bar{m}_2 := r \cdot f'\). These definitions, however, could be immediately included into the field definitions without loss of efficiency.

The proof shows that this construction satisfies the universal property. We omit this part of the proof here. Figures 6b and 6c summarize how to construct \(u\) from two morphisms \(g'\) and \(f'\) with \(f' \cdot m_2 = g' \cdot m_1\).
Checking the commutativity also tests that $\text{dom}(\bar{g}') = O_1$ and $\text{dom}(\bar{f}') = O_2$.

Now, we have generic data types describing initial object, coproducts, coequalizers, and pushouts. Analogously, we can write default implementations for other colimit constructions such as the ternary coproduct of three objects.

### 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:

\[
\begin{array}{ccc}
A & \xrightarrow{f_1} & B \\
\downarrow{g} & & \downarrow{\bar{g}_1} \\
C & \xrightarrow{f_3} & D
\end{array}
\quad\begin{array}{ccc}
& & \\
\bar{g}_2 & \xrightarrow{f_2} & B'
\end{array}
\]

(a) If both the left-hand subdiagram $\bar{g}_1 \cdot f_1 = f_1 \cdot g$ and the right-hand subdiagram $\bar{g}_2 \cdot f_2 = f_2 \cdot \bar{g}_1$ are pushout diagrams, then so is the outer diagram $\bar{g}_2 \cdot f_2 \cdot f_1 = f_2 \cdot f_1 \cdot g$.

(b) Conversely, if the left-hand subdiagram and the outer diagram are pushout diagrams, then the right-hand subdiagram is a pushout diagram, too.

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.\(^3\)

\(^3\) More precisely, the morphisms must be identical up to isomorphism. But, checking this is too expensive to be implemented here.
composePushouts c po1 po2
    = if poMor2b po1 /= poMor2 po2 then
        error "composePushouts: do not fit together" else
    If this condition is satisfied, we can copy the components of the resulting pushout
diagram immediately from the given ones:

    Pushout {poInt = poInt po1,
        poObj1 = poObj1 po2,
        poObj2 = poObj2 po1,
        poObj = poObj po2,
        poMor1 = f2f1,
        poMor2 = poMor2 po1,
        poMor1b = f2f1b,
        poMor2b = poMor2b po2,
        poUniv = u
    }

    where
    f2f1 = composeWithoutTest c (poMor1 po2) (poMor1 po1)
    f2f1b = composeWithoutTest c (poMor1b po2) (poMor1b po1)

    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:

    u q p =
        if (compose c q f2f1) /= (compose c p (poMor2 po1))
        then error "poUniv (compose): not commutative"
        else
            poUniv po2 q h
            where
            h = poUniv po1 (compose c q (poMor1 po2)) p

    The second part of the lemma can be implemented analogously.
3 Special Categories

Haskell classes are classes of types, not classes of objects.\(^4\) Therefore, we have to define a data type the objects of which are categories. This is done in a separate module:

```haskell
module SpecCategories where

import Category
import Colimits

If we should implement the limit constructions, we must import this module, too.

The data type of categories must provide the user with the operations defined in the category. This includes the operations mentioned in the definition of a category as well as the colimit operations.

The data type of categories depends on the data types of objects and of morphisms:

```haskell
    data (Obj o, Mor m) =>
    Cat o m =
    Cat {
        thisDom:: m -> o,
        thisCodom:: m -> o,
        thisCompose:: m -> m -> m,
        thisIdent:: o -> m,
    }
```

We provide this data type with the operations defined for every category:

```haskell
    thisInitial:: Initial o m,
    thisCoproduct:: o -> o -> Coproduct o m,
    thisCoequalizer:: m -> m -> Coequalizer o m
}
```

Of course, we do not need the operations for which default implementations exist.

We have implemented \texttt{Cat} by an algebraic data type with labeled fields. We benefit from these field labels in two ways. First, they give us mnemonic access to the operations. Furthermore, this technique allows us to omit some components,

\[^4\text{This is an important difference between Haskell and object-oriented programming.}\]
e.g., because either the operation is not defined in a category or we have not yet implemented it.

This data type must be made an instance both of the class `Category` and of the class `Colimits`:

```haskell
instance Category Cat where
  dom = thisDom
  codom = thisCodom
  composeWithoutTest = thisCompose
  ident = thisIdent

instance CatWithColimits Cat where
  initial = thisInitial
  coproduct = thisCoproduct
  coequalizer = thisCoequalizer
```

*Remark:* From a systematic point of view, this definition should be placed in the module `Category`. But that solution makes the modules `Category` and `Colimits` recursive. Recursive modules, however, are not supported by the interpreter Hugs, which is more convenient than the compiler Ghc.

## 4 The Category of Sets

### 4.1 Defining the Interface

The concrete implementation of the category of sets is an object of type `Cat` as defined in the module `SpecCategories`. We arrange the definition of this object together with all its properties in a module `CatSet`:

```haskell
module CatSet (module Category, module Colimits,
  ObjType, ArrowType,
  Set (constructSet), Map (consMapFromPairList),
  catSet, SetObj (SetObj), SetMor (setArrow),
  consSetMor) where
  
  This export list will be explained later. We have to import `SpecCategories` and the modules it is based on:

  import Category
  import Colimits
  import SpecCategories
```
If we add an export clause to the module `SpecCategories`, it is not necessary to import `Category` and `Colimits` here. The present version, however, makes clear which properties of `Set` are already implemented.

Defining the category of sets and its methods, we need a suitable representation of sets and maps:

```haskell
import Set
import SetMap
import CatSetImpl
```

In the modules `Set` and `SetMap`, we have implemented some functions manipulating sets and maps. The functionalities imitate the Java interfaces given by the API Specification of Java 2 Standard Edition [5].

The module `CatSetImpl` is used as a link between the implementation of sets and of maps and the categorical data types. It defines concrete versions of the types `ObjType` and `ArrowType`:

```haskell
module CatSetImpl (module CatSetImpl) where

import SetSeq

type ObjType = SetSeq String
type ArrowType = SetSeq (SetMapEl String String)
```

These types are implementations of `Set` and of `SetMap`, respectively. Thus, we can apply the set and map methods to objects of these types.

Putting these definitions in a separate module makes it simpler to switch between different versions without altering the main modules. The present version uses alphanumeric strings to identify set elements. Alternatively, we have tested a version using integers.

The export list deserves some more discussion. Of course, we export all the categorical notions and constructions, i.e., all the definitions imported from the modules `Category` and `Colimits`. Furthermore, the types of objects and morphisms must be known to modules working with the category of sets. Therefore, we must include `ObjType` and `ArrowType` into the export list, too. But, we do not re-export all the definitions imported from `Set` and `SetMap`, preventing client modules from accessing the implementation details. We provide the users only with `constructSet` and `consMapFromPairList` [5]. Thus, they are able to construct sets and mappings, but not to alter them. The last two lines of the export list will be discussed below.

As we have already mentioned, the category of sets is a programming language object of type `Cat o m`, where `o` is the type of the objects and `m` is the type of the morphisms:

[5] Unfortunately, the Haskell syntax does not prevent the client modules from explicitly importing `Set` and `SetMap`. 

catSet = Cat \{thisDom = setDom, 
thisCodom = setCodom, 
thisCompose = setCompose, 
thisIdent = setIdent, 
thisInitial = setInitial, 
thisCoproduct = setCoproduct, 
thisCoequalizer = setCoequalizer \}

The remaining part of this module is concerned with specifying the functions on the right-hand sides of these definitions.

### 4.2 Objects and Morphisms

In order to build the category of sets, we first introduce a new data type distinguishing usual sets from categorical set objects, and we make it instances of `Obj` and `Show`:

```hs
newtype SetObj = SetObj {toSet:: ObjType} deriving Eq

instance Obj SetObj

instance Show SetObj where
    show o = show (toSet o)
```

The constructor `SetObj` and the field label `toSet` act as transfer functions between a set and its view as a categorical object. This allows us to carefully distinguish between both views.

Next, we introduce the type of set morphisms:

```hs
data SetMor = SetMor \{setDom:: SetObj, 
    setArrow:: ArrowType, 
    setCodom:: SetObj\} deriving Eq
```

Making set morphisms an instance of `Mor`, we have to determine that in `Set`, monomorphisms are injections and epimorphisms are surjections:

```hs
instance Mor SetMor where
    isMonomorphic m = card (keySet (setArrow m))
        == card (valueSet (setArrow m))
    isEpimorphic m = card (valueSet (setArrow m))
        == card (toSet (setCodom m))
```
Although we can now construct set morphisms from two set objects and a mapping using the constructor `SetMor`, this is not satisfying since there is no check that the domain and the codomain objects (as sets) agree with the key set and the value set of the mapping, respectively. Therefore, we introduce a method constructing morphisms together with this check:

```
consSetMor o1 f o2
  = if (toSet o1 /= keySet f)
      then error "consSetMor: incorrect domain"
      else if subset (valueSet f) (toSet o2) then
        SetMor {setDom = o1,
                 setArrow = f,
                 setCodom = o2}
      else error "consSetMor: incorrect codomain"
```

Now, we resume discussion of the export list. We do not want that client modules use the constructor `SetMor` avoiding the compatibility check. Therefore, we do not export all the definitions of the module `CatSet`, but take advantage of selective export. First of all, we must export the object `catSet`. Furthermore, client modules need to know the data types `SetObj` and `SetMor`. No problems arise from exporting the data type `SetObj` together with its constructor, whereas the data type `SetMor` is exported without its constructor. Instead, we provide the client modules with the method `consSetMor`. Finally, we allow other modules to access the set mapping a morphism is based on by exporting the field `setArrow`. (E.g., we need this component to construct graph morphisms.)

The data type `SetMor` implicitly defines the methods `setDom` and `setCodom`. The definition of a category furthermore requires that we describe how to compose two morphisms and how to construct the identity morphism of an object:

```
setCompose g f
  = SetMor {setDom = setDom f,
            setArrow = mapCompose (setArrow g) (setArrow f),
            setCodom = setCodom g}
```

Of course, we can use the constructor `SetMor` here without the compatibility check, since the formal definition ensures correctness.

```
setIdent o
  = SetMor {setDom = o,
            setArrow = id,
            setCodom = o}
```

where `id = consMapFromPairList [(e,e) | e <- toList(toSet o)]`

The function `toList` converts a set into a list containing the same elements regardless of how the set is implemented.
4.3 Initial Object and Coproduct

We need to implement only the basic colimits, i.e., the initial object, the coproduct, and the coequalizer. Then, we can construct all the other colimits, e.g., the pushout, taking advantage of the generic methods.

**Lemma 8 (Initial object in $\mathcal{S}et$):** The initial object in $\mathcal{S}et$ is the empty set.

\[
\text{setInitial} = \text{Initial} \{ \text{initObj} = \text{obj}, \\
\text{initUniv} = \text{u} \}
\]

\[
\text{where } \text{obj} = \text{SetObj emptySet}
\]

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

\[
\text{u } o' = \text{SetMor} \{ \text{setDom} = \text{obj}, \\
\text{setArrow} = \text{emptyMap}, \\
\text{setCodom} = o' \}
\]

\[
\text{emptyMap} = \text{consMapFromPairList} []
\]

Implementing the coproduct, we refer to the properties proved in [4, Chapter 2]:

**Lemma 9 (Coproduct in $\mathcal{S}et$):** The coproduct in $\mathcal{S}et$ is the disjoint union together with the natural injections.

Since we can not assume that the given sets are disjoint, we introduce an auxiliary function labeling the elements of the first set with 1 and those of the other set with 2.

\[
\text{markSetElmts} :: \text{ObjType} -> \text{Int} -> \text{ArrowType}
\]

The resulting function maps the original identifier to the new one:

\[
\text{markSetElmts} \text{s i} \\
= \text{if isEmptySet s then consMapFromPairList []} \\
else \text{put e (mark e i) (markSetElmts (remove e s) i)} \\
\text{where e = pick s}
\]

The function `mark` depends on the way we identify the elements of a set and therefore, it must be included in the module `CatSetImpl`. In case of alphanumeric strings, we append the characters 1 or 2, respectively.
setCoproduct o1 o2
    = Coproduct {cpObj1 = o1,
                  cpObj2 = o2,
                  cpObj = obj,
                  cpMor1 = consSetMor o1 map1 obj,
                  cpMor2 = consSetMor o2 map2 obj,
                  cpUniv = coprodUniv }

Due to making the given sets disjoint, we get the coproduct set as the usual union of the marked sets:

    where obj = SetObj (union s1marked s2marked)
      s1marked = valueSet map1
      s2marked = valueSet map2
      map1 = markSetElmts (toSet o1) 1
      map2 = markSetElmts (toSet o2) 2

To implement the universal property, we consider two arbitrary mappings $f'_1: O_1 \rightarrow C'$ and $f'_2: O_2 \rightarrow C'$,

$$\text{coprodUniv } f_1 f_2$$

    = if (setDom f1) /= o1
        || (setDom f2) /= o2 then
          error "Domains don't match"
        else if (setCodom f1) /= (setCodom f2)
          then error "Codomains don't match"
          else SetMor {setDom = obj,
                        setArrow = u,
                        setCodom = setCodom f1
                      }

and we define

$$u(c) := \begin{cases} f'_1(x) & \text{if } c \in m_1[O_1] \land c = m_1(x) \\ f'_2(x) & \text{if } c \in m_2[O_2] \land c = m_2(x) \end{cases}$$

where

$$u = \text{consMapFromPairList}$$

    ([(get x map1, get x (setArrow f1))
        | x <- toList((toSet o1)) ] ++
     [(get x map2, get x (setArrow f2))
        | x <- toList((toSet o2)) ]
  )
4.4 Coequalizer

The implementation of the coequalizer in \( \text{Set} \) is a little bit more sophisticated, since we have to construct equivalence classes:

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

(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.,

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

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

```plaintext
cetCode m1 m2
  = if (dom catSet m1 /= dom catSet m2)
    || (codom catSet m1 /= codom catSet m2)
    then error "SetCoequalizer: Morphisms not parallel"
  else Coequalizer {ceqObj1 = dom catSet m1,
                    ceqObj2 = codom catSet m1,
                    ceqObj  = o3,
                    ceqMor1 = m1,
                    ceqMor2 = m2,
                    ceqMor  = consSetMor o2 f o3,
                    ceqUniv = univ
                }
```

If the morphisms \( m_1 \) and \( m_2 \) are parallel, we can construct the coequalizer:

```plaintext
else Coequalizer {ceqObj1 = dom catSet m1,
                  ceqObj2 = codom catSet m1,
                  ceqObj  = o3,
                  ceqMor1 = m1,
                  ceqMor2 = m2,
                  ceqMor  = consSetMor o2 f o3,
                  ceqUniv = univ
                }
```

First, we define some abbreviations:

```plaintext
where f1 = setArrow m1
f2 = setArrow m2
s1 = toSet (dom catSet m1)
o2 = codom catSet m1
```

The main point is constructing the equivalence classes as defined in Lemma 10(b). We start with a set of classes each of which contains exactly one element of the second set. Then, we run through the set \( s1 \) putting together the classes containing \( m_1(e) \) and \( m_2(e) \):
startClasses = makeSingletons (toSet o2)
equivClasses = closure s1 startClasses

closure r classes
  = if isEmptySet r then classes
    else closure
      (remove e r)
      (findUnite (get e f1) (get e f2) classes)
    where e = pick r

To construct the coequalizer object, we represent each equivalence class by its first element, i.e., all the elements of an equivalence class are mapped to its first element.

f = consMapFromPairList
  (reduce (toList equivClasses))

Here, equivClasses is a set of sets, reduce is applied to a list of sets:

reduce [] = []
reduce (s:r) = (mapToFirst (toList s)) ++ reduce r

Function reduce produces a list of pairs the first components of which run through all the subsets. The second component of each pair determines the head element of the subset its first component is contained in.

mapToFirst [] = []
mapToFirst (e:r) = [(e,e)] ++ [(e',e) | e' <- r]
o3 = SetObj (valueSet f)

The reduced set yields the coequalizer object, and the map f is the arrow of the coequalizer morphism ceqMor.

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 \( f(y) \mapsto q'(y) \) to the map.

univ q
  = if compose catSet q m1 /= compose catSet q m2
    then error "SetCoequal: q m1 /= q m2"
    else consSetMor o3 uMap (codom catSet q)
  where mMap = setArrow q
    uMap = makeUniv (toSet o2)
    makeUniv s
      = if isEmptySet s then consMap []
        else put (get y f) (get y mMap)
          (makeUniv (remove y s))
        where y = pick s
Some of these pairs 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 put implicitly removes these duplicates.

5 The Category of Graphs

The category of graphs is based on the category of sets. We import this module
as well as the categorical modules:

module CatGraph (module Category, module Colimits,
    module CatSet,
catGraph, consGraphObj, consGraphMor) where

    import Category
    import Colimits
    import SpecCategories
    import CatSet

Of course, we cannot import the implementation details that are not exported
by the module CatSet. Again, we restrict export to safe operations.

5.1 Graph Objects

We use the definitions we have explained in full detail in [4, Chapter 2].

**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.

This definition can be immediately translated into a data type:

```haskell
    data GraphObj = GraphObj {edges:: SetObj,
                               nodes:: SetObj,
                               source:: SetMor,
                               target:: SetMor }
```

This data type must be made an instance of \text{Obj} assuming that it is also an
instance of \text{Eq}:

```haskell
    instance Obj GraphObj

    instance Eq GraphObj where
        g1 == g2 = (source g1) == (source g2)
        && (target g1) == (target g2)
```
We can omit the explicit check that the set of edges and the set of nodes of \( g_1 \) equals the set of edges and the set of nodes of \( g_2 \), respectively, since these tests are performed by the equality test on the morphisms.

The constructor does not check that \( \text{source} \) and \( \text{target} \) are set morphisms from \( \text{edges} \) to \( \text{nodes} \). As in the case of sets, a special method is provided constructing graphs and checking this condition. Again, only this method can be accessed from outside the module:

```haskell
consGraphObj s t
  = if (dom catSet s) /= (dom catSet t)
    || (codom catSet s) /= (codom catSet t) then
      error "consGraphObj: source/target not parallel"
    else GraphObj {edges = dom catSet s,
      nodes = codom catSet s,
      source = s,
      target = t }
```

It is sufficient to supply this method with the morphisms defining source and target. The set of edges and the set of nodes are then defined implicitly. What about isolated nodes that are neither the source nor the target of an edge? Please, take into consideration that the parameters \( s \) and \( t \) are not mappings, but set morphisms from \( E \) to \( V \); thus, their codomains include all the nodes.

### 5.2 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 condition can be illustrated graphically:

\[
\begin{array}{c}
E_G \xrightarrow{s_G} V_G & \quad & E_G \xrightarrow{t_G} V_G \\
\downarrow_{f_E} & = & \downarrow_{f_V} \quad \downarrow_{f_E} & = & \downarrow_{f_V} \\
E_H \xrightarrow{s_H} V_H & & E_H \xrightarrow{t_H} V_H
\end{array}
\]

The data type includes the graphs \( G \) and \( H \), the set morphism \( f_E \) between the edges, and the set morphism \( f_V \) between the nodes:

\( E_G \) 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.
data GraphMor = GraphMor {graphDom :: GraphObj, 
edgeArrow :: SetMor, 
nodeArrow :: SetMor, 
graphCodom :: GraphObj } deriving Eq

In this formulation, we take advantage of Haskell's default instantiation defining the equality test componentwise, although it is not efficient. The equality test on the second component as well as the test on the third component examine $E_G = E_H$ and $V_G = V_H$, but do not check equality of the source and target mappings. Therefore, we also check the first and the last component. Of course, you may improve the efficiency by giving an explicit instantiation considering the details of graph objects and of set morphisms.

The definitions of monomorphisms and epimorphisms are derived from the category of sets:

instance Mor GraphMor where
  isMonomorphic m = isMonomorphic (edgeArrow m)
                    && isMonomorphic (nodeArrow m)
  isEpimorphic m  = isEpimorphic (edgeArrow m)
                    && isEpimorphic (nodeArrow m)

As usual, we do not export the constructor GraphMor, but a method that constructs a graph morphism only after checking the commutativity of the diagrams given in Definition 12:

consGraphMor g fe fv h
  = if (compose catSet fv (source g)) /= (compose catSet (source h) fe) then
        error "consGraphMor: Source diagram not commutative"
    else if (compose catSet fv (target g)) /= (compose catSet (target h) fe) then
        error "consGraphMor: Target diagram not commutative"
    else GraphMor {graphDom = g,
                   edgeArrow = fe,
                   nodeArrow = fv,
                   graphCodom = h }

We do not test explicitly that $f_E$ is a morphism from $E_G$ to $E_H$. This is done implicitly by the (in-)equality tests at the beginning of this method. E.g., the compose on the right-hand side of the last test ensures that the codomain of $f_E$ is the domain of $t_H$, i.e., $E_H$. On the other hand, equality of the two composed
morphisms requires that both morphisms start at the same object, i.e, \( f_E \) starts at the same object as \( t_G \).

May be that later versions require `edgeArrow` and `nodeArrow` to be exported, too.

### 5.3 Category of Graphs

**Theorem 13 (Category \( \mathcal{G}raph \))**: The class of graphs and the class of graph morphisms together with componentwise composition: \( g \cdot f = (g_E \cdot f_E, g_V \cdot f_V) \) constitute the category \( \mathcal{G}raph \).

As in the case of \( \text{catSet} \), we have to define the programming language object \( \text{catGraph} \) and its components:

\[
\text{catGraph} = \text{Cat} \left\{ \text{thisDom} = \text{graphDom}, \right.
\left. \text{thisCodom} = \text{graphCodom}, \right.
\left. \text{thisCompose} = \text{graphCompose}, \right.
\left. \text{thisIdent} = \text{graphIdent}, \right.
\left. \text{thisInitial} = \text{graphInitial}, \right.
\left. \text{thisCoproduct} = \text{graphCoproduct}, \right.
\left. \text{thisCoequalizer} = \text{graphCoequalizer} \right\}
\]

The theorem states that composition of graph morphisms is defined component-wise:

\[
\text{graphCompose } g \ f
= \text{GraphMor} \left\{ \text{graphDom} = \text{graphDom } f, \right.
\left. \text{edgeArrow} = \text{compose } \text{catSet} (\text{edgeArrow } g) (\text{edgeArrow } f), \right.
\left. \text{nodeArrow} = \text{compose } \text{catSet} (\text{nodeArrow } g) (\text{nodeArrow } f), \right.
\left. \text{graphCodom} = \text{graphCodom } g \right\}
\]

The proof of Theorem 13 shows that the identities are also defined component-wise:

\[
\text{graphIdent } o
= \text{GraphMor} \left\{ \text{graphDom} = o, \right.
\left. \text{edgeArrow} = \text{ident } \text{catSet} (\text{edges } o), \right.
\left. \text{nodeArrow} = \text{ident } \text{catSet} (\text{nodes } o), \right.
\left. \text{graphCodom} = o \right\}
\]
5.4 Colimits in $\mathcal{G}raph$

All the colimit constructions in the category of graphs are based on constructing the colimits for edges and nodes in $\mathcal{S}et$, 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:

$$\text{graphCoproduct } o_1 \ o_2$$
$$= \text{Coproduct } \{ cpObj1 = o_1, \$$
$$cpObj2 = o_2, \$$
$$cpObj = obj, \$$
$$cpMor1 = \text{consGraphMor } o_1 \ f_1 e \ f_1 v \ \text{obj}, \$$
$$cpMor2 = \text{consGraphMor } o_2 \ f_2 e \ f_2 v \ \text{obj}, \$$
$$cpUniv = \text{univ} \}$$

where $cpEdges = \text{coproduct catSet } (\text{edges } o_1) \ (\text{edges } o_2)$
$cpNodes = \text{coproduct catSet } (\text{nodes } o_1) \ (\text{nodes } o_2)$
$f_1 e = cpMor1 \ cpEdges$
$f_2 e = cpMor2 \ cpEdges$
$f_1 v = cpMor1 \ cpNodes$
$f_2 v = cpMor2 \ cpNodes$

Then, the coproduct property for the edges can be used to define the source and the target functions of the coproduct object in $\mathcal{G}raph$:

Implementing the universal property, we have to ensure first that the given morphisms satisfy the condition of Definition 2:
univ fb1 fb2
  = if (graphDom fb1) /= o1
    || (graphDom fb2) /= o2 then
    error "Coproduct-Univ: Domains don’t match"
  else
    if (graphCodom fb1) /= (graphCodom fb2) then
    error "Coproduct-Univ: Codomains don’t match"
  else

Then, the resulting morphism is defined componentwise:

\[
\text{GraphMor} \{ \text{graphDom} = \text{obj}, \\
\text{edgeArrow} = \text{cpUniv} \text{cpEdges} \\
  \quad (\text{edgeArrow fb1}) \\
  \quad (\text{edgeArrow fb2}), \\
\text{nodeArrow} = \text{cpUniv} \text{cpNodes} \\
  \quad (\text{nodeArrow fb1}) \\
  \quad (\text{nodeArrow fb2}), \\
\text{graphCodom} = \text{graphCodom fb1} \\
\}
\]

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

Analogously, we construct the coequalizer in \textit{Graph} for edges and nodes separately. We first check whether the morphisms are parallel:

\[
\text{graphCoequalizer} m1 m2
  = \text{if} \ (\text{dom catGraph m1} /= \text{dom catGraph m2})
  || (\text{codom catGraph m1} /= \text{codom catGraph m2})
  \text{then} \text{error "SetCoequalizer: Morphisms not parallel"}
\]

Then, some information is immediately copied into the result:

\[
\text{else Coequalizer} \{ \text{ceqObj1} = \text{dom catGraph m1}, \\
\text{ceqObj2} = \text{dom catGraph m2}, \\
\text{ceqObj} = \text{obj}, \\
\text{ceqMor1} = m1, \\
\text{ceqMor2} = m2, \\
\text{ceqMor} = m, \\
\text{ceqUniv} = \text{univ} \\
\}
\]

Separately considering edges and nodes:
where
\[ \text{ceqEdges} = \text{coequalizer } \text{catSet} \]
\[ (\text{edgeArrow } m_1) \ (\text{edgeArrow } m_2) \]
\[ \text{ceqNodes} = \text{coequalizer } \text{catSet} \]
\[ (\text{nodeArrow } m_1) \ (\text{nodeArrow } m_2) \]
yields the edges and the nodes of the coequalizer object as well as the mappings
from the second object to the result:

\[
\text{obj} = \text{GraphObj} \begin{cases} 
\text{edges} = \text{ceqObj } \text{ceqEdges}, \\
\text{nodes} = \text{ceqObj } \text{ceqNodes}, \\
\text{source} = \text{src}, \\
\text{target} = \text{tgt} 
\end{cases}
\]
\[ o_2 = \text{codom } \text{catGraph } m_1 \]
\[ m = \text{consGraphMor } o_2 \ (\text{ceqMor } \text{ceqEdges}) \]
\[ (\text{ceqMor } \text{ceqNodes}) \ \text{obj} \]

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:

\[
E_1 \xrightarrow{m_{E_1}} E_2 \xrightarrow{m_E} E \\
V_1 \xrightarrow{m_{V_1}} V_2 \xrightarrow{m_V} V
\]

\[
\text{src} = \text{ceqUniv } \text{ceqEdges} \]
\[ (\text{compose } \text{catSet} \ (\text{ceqMor } \text{ceqNodes}) \]
\[ (\text{source } o_2)) \]
\[ \text{tgt} = \text{ceqUniv } \text{ceqEdges} \]
\[ (\text{compose } \text{catSet} \ (\text{ceqMor } \text{ceqNodes}) \]
\[ (\text{target } o_2)) \]

Finally, we need the universal property:
\[
\text{univ } q \\
= \text{if } \text{compose } \text{catGraph } q \text{ m1 }\neq \text{compose } \text{catGraph } q \text{ m2} \\
\text{then error "GraphCoequal: } q \text{ m1 }\neq q \text{ m2" } \\
\text{else GraphMor } \\
\{ \text{graphDom = obj,} \\
\text{edgeArrow = ceqUniv ceqEdges (edgeArrow } q), \\
\text{nodeArrow = ceqUniv ceqNodes (nodeArrow } q), \\
\text{graphCodom = codom catGraph } q \\
\} 
\]

Since the graphs involved in the construction are given, it is not necessary to consider source and target functions. Furthermore, we can use the constructor \text{GraphMor}, i.e., we do not check the commutativity condition of Definition 12, because it is a consequence of the construction.

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

\[
\text{graphInitial } \\
= \text{Initial } \{ \text{initObj = obj,} \\
\text{initUniv = u } \\
\}
\]

where \text{baseObj} = \text{initial } \text{catSet} \\
\text{baseInit = initObj baseObj} \\
\text{obj = GraphObj } \{ \text{edges = initObj baseObj,} \\
\text{nodes = initObj baseObj,} \\
\text{source = ident catSet baseInit,} \\
\text{target = ident catSet baseInit } \\
\}

The universal property is just as simple:

\[
\text{baseUniv = initUniv baseObj } \\
\text{u o' = GraphMor } \{ \text{graphDom = obj,} \\
\text{edgeArrow = baseUniv (edges o'),} \\
\text{nodeArrow = baseUniv (nodes o'),} \\
\text{graphCodom = o' } \\
\}
\]

This completes the implementation of the category of graphs. In [4, 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.
6 Conclusion

An advantage of Haskell, which immediately leaps into the eye, is multiple inheritance. In this draft, we have defined only the colimit constructions. Analogously, we can define the limit constructions in a class \texttt{Limits}, and then make the category of sets a subclass of both classes. Remember that this is not possible in Java; we have to use interfaces instead, but an interface must not implement methods. Therefore, we have to define the general constructions separately as factory methods that each category must explicitly call in its constructor.

Another advantage is that functions are first-class objects. This allows us to use the function identifiers and omit the arguments. We have taken advantage of this concept, e.g., in defining the universal functions in \texttt{CatSet} and in \texttt{CatGraph}.

Last but not least, we mention the fact that Haskell does not impose much restrictions on the order of definitions. Using literate programming, i.e., combining \LaTeX{} and Haskell, we can take much advantage of this freedom by arranging the definitions such that easy understanding is supported. The \LaTeX{} source of the present document is valid Haskell code.

Until now, we have uncovered two disadvantages. Field labels used in one data type must not be re-used in another. This forces us to find different identifiers to denote the components of different colimits. Another disadvantage is that we can not prevent client modules from importing the modules defining the details of set implementation.

Our first impression is that Haskell is well-suited to implement a generic concept such as categorical constructions.

References

5. Sun Microsystems: \textit{API Specification}, [http://java.sun.com/j2se/1.5.0/docs/api/](http://java.sun.com/j2se/1.5.0/docs/api/) (Link checked on Jan. 23rd, 2007)