Exploring Game Comonads in Haskell

Alastair Flynn

Honour School of Computer Science - Part C

Trinity Term, 2021

Contents

Abstract

Abramsky-Shah's paper *Relating Structure and Power* gives a categorical characterization of key concepts from finite model theory. The paper links two disjoint areas of logic in computer science: categorical semantics and finite model theory. In finite model theory model comparison games are used to establish logical equivalence between models. Two of these, the Ehrenfeucht-Fraïssé and Pebbling games, are given categorical semantics in the paper. The coKleisli morphisms of two indexed families of comonads on the category of relational structures are used to give syntax-free characterizations of the games.

In this paper we give a programmatic representation of these comonads in Haskell, a language known for its integration of categorical concepts. We begin by giving a framework for implementing arbitrary categories in Haskell. Implementations of these comonads over the category of relational structures are then given with the signature containing one predicate of arity 2. The objects of this category are isomorphic to graphs. We construct a selection of functions that allow for proofs of inexpressibility in particular logics by showing the existence of certain coKleisli morphisms. Several examples of inexpressibility proofs are included using this technique. Numerous other useful functions are also given.

Furthermore, rank k coalgebras of the Ehrenfeucht-Fraïssé comonad correspond to the combinatorial parameter *tree depth*. In this paper we give conversions between the coalgebras of rank k and the tree depth decompositions of depth k of graphs in the category.

Chapter 1

Introduction

In theoretical computer science the study of *structure*, *i.e.* semantics and compositionality, and the study of *power*, *i.e.* expressiveness and complexity, have always been two somewhat disparate fields. This changed with Abramsky's seminal paper *The Pebbling Comonad in Finite Model Theory* [1], which established a new link between these two fields. This was then expanded upon in *Relating Structure and Power* by Abramsky-Shah [2]. These two papers use categorical concepts (*structure*) to captured key components from finite model theory (*power*). This opened up a whole new way of giving proofs in finite model theory alongside providing a new way of expressing the two combinatorial parameters: tree width and tree depth. As of yet, this has not been explored programmatically.

Model Theory

Model theory is based on relational structures (models) and the homomorphisms (structure preserving maps) between them [16]. These models and homomorphisms also play a fundamental role in constraint satisfaction and database theory [5, 9]. If there exists a homomorphism from one model to another this is equivalent to saying the target model satisfies the same positive sentences as the source. A model is a set of relations over a signature. A signature is called a *schema* in database theory.

An important question in model theory is whether two models are equivalent under a given logic. Due to an major result by Ehrenfeucht-Fraïssé [8, 10], for a selection of logics a winning strategy in a certain model comparison game corresponds to a proof that the two models are equivalent. These model comparison games are played between two players: the Spoiler and the Duplicator. The Spoiler wants to show that the two models are the different, the Duplicator is trying to show they are the same. If the Duplicator has a winning strategy, then the two models are the same.

These games are bounded by a resource parameter k. In [2], two of these games, the Ehrenfeucht-Fraïssé game and the Pebbling game are given categorical semantics. In the first, k is the maximum number of rounds and in the second it is a limit on the number of pebbles.

The Link

The categorical structures that capture these model comparison games are families of comonads indexed by the resource parameter k. Comonads are the categorical dual of monads and similarly elude an intuitive explanation. In Haskell, comonads provide a useful design pattern, capturing notions of context dependence [19]. For the game comonads we may think of the context as the resource parameter k.

In finite model theory, the standard way to show that two models are equivalent is to prove Duplicator's strategy always results in a win. Using the comonads, we can instead show that, in the category of relational structures, we have morphisms $\mathbb{C}_k\mathscr{A} \to \mathscr{B}$ and $\mathbb{C}_k\mathscr{B} \to \mathscr{A}$ where $\mathbb{C}_k$ is an indexed game comonad. This is due to results in [2].

Constructing these morphisms programmatically, in a type safe way, reduces a proof that two models are equivalent to checking that we have the relevant valid morphisms.

Many well known functional programming languages are built around category theory [12, 13], most notably Haskell. This makes it the obvious choice to implement the comonads and build a framework for checking the morphisms.

To implement the comonads we first need an implementation of the category they act over. **Hask** is a category with Haskell types as object and Haskell functions as morphisms[1]. After we implemented subcategories of **Hask** we then attempted to build a representation of $\mathcal{R}(\sigma)$ which can be viewed as a subcategory of **Hask**. Unfortunately, without dependent types, which Haskell does not yet have a standard way of implementing, this is very difficult. Instead, we choose to provide an implementation of the category of relational structures with the signature that contains one predicate of arity 2. This is isomorphic to the category of graphs. In finite model theory it is often said all interesting things can be done with graphs, hence why this category was chosen. Other design decisions are discussed in chapter 4.

We implement the comonads over $\mathcal{R}(\sigma_G)$ with a range of useful functions summarized in chapter 5. The full implementation is explained in chapter 6, and in chapter 7 we give several examples that use our code to give proofs from finite model theory.

The full code listing is given in the appendix. It can also be found on GitHub[2].

Combinatorial Parameters

Tree width and tree depth are properties of graphs. We can think of tree width as measuring how far the graph is from being a tree and tree depth as measuring how far it is from being a star. These parameters have relevance in computational complexity where algorithmic problems that are NP-complete for arbitrary graphs can be solved efficiently for graphs with bounded tree width/depth [4, 14].

A comonad is linked with the concept of a coalgebras, that is, morphisms $\mathscr{A} \to \mathbb{C}_k\mathscr{A}$ which satisfy certain rules. For the Ehrenfeucht-Fraïssé comonad these coalgebras are isomorphic to tree depth decompositions of $\mathscr{A}$ and for the Pebbling comonad, they are isomorphic to tree width decompositions.

In Haskell, we give an implementation of the isomorphism between the coalgebras of the Ehrenfeucht-Fraïssé comonad and tree depth decompositions of the models.

Contributions

The implementation provided with this paper is designed primarily as a starting point for future research. In the conclusion we outline the shortcomings of the implementation and how it could be extended.

The initial aim of this project was to provide a Haskell library which would allow researchers to use the comonads to construct morphisms in the category along with all the features discussed above and more. Due to time constraints the work was not published as a library; it remains a work in progress.

[1] Whether this actually forms a valid category is hotly debated (see section 4.3). For the purposes of this paper it suffices.

[2] `https://github.com/throwaway178/GraphComonads`

The primary contributions are the implementation, in Haskell, of the following:

- A framework for building subcategories of **Hask**
- Ehrenfeucht-Fraïssé and Pebbling comonad over the category of graphs
- Proof checkers for the logics associated with the Ehrenfeucht-Fraïssé comonad
- Examples of proofs using the comonads
- Conversions between Ehrenfeucht-Fraïssé coalgebras and tree depth decompositions

Chapter 2

Background

To fully grasp the power of the game comonads implemented in Haskell it helps to first understand the two fields it unites. In this chapter we review both the fundamental and relevant concepts from the fields of category theory and finite model theory.

2.1 Category Theory

Category theory provides a universal approach to formalizing various areas of mathematics and computer science. We illustrate the concepts introduced using examples from the category of graphs, **Graph**. We take a "graph" to mean a directed graph $G = (V, E)$ where V is a set of vertices and $E \subseteq V^2$. A graph homomorphism $h : G_1 \to G_2$ is a function $h : V_1 \to V_2$ such that for every $(v_1, v_2) \in E_1$ we have $(h(v_1)), h(v_2)) \in E_2$.

Categories

Category theory is based on the fundamental concept of a category. A category $\mathcal{C}$ consists of:

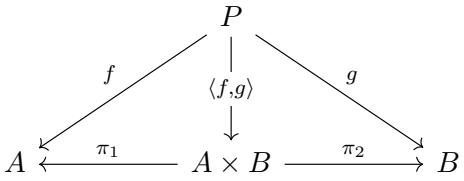
- A collection of objects $Ob(\mathcal{C})$
- A collection of morphisms $f : A \to B$ for all $A, B \in Ob(\mathcal{C})$. These satisfy the following conditions:
  - Any two morphisms, $f : A \to B$, $g : B \to C$, can be composed using the composition map $g \circ f : A \to C$. The map is associative, that is, $(f \circ g) \circ h = f \circ (g \circ h)$.
  - For any object $A \in Ob(\mathcal{C})$ there exists an identity object, id_A, such that $id_A \circ f = id_A = f \circ id_A$.

In **Graph** the objects are graphs and the morphisms are graph homomorphisms. These clearly satisfy the conditions.

Products/Coproducts

A a category $\mathcal{C}$ has *products* if for every pair of morphisms, $f : P \to A$, $g : P \to B$, there exists an object $A \times B \in Ob(\mathcal{C})$ and a unique morphism $\langle f, g \rangle : P \to A \times B$ along

with projectors $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that the following diagram commutes[1]:

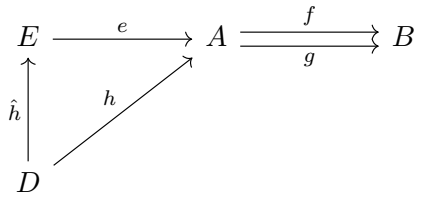


This effectively says that any object with morphisms to A and B must factor through a product object $A \times B$. The general term for this property is *universality*.

We can get the definition of coproducts by reversing all the morphisms in the diagram. The coproduct of A and B is denoted $A + B$. In **Graph**, the vertices of the product graph $G_1 \times G_2$ are $V_1 \times V_2$ and there is an edge between two pairs iff both the first and second elements are connected in the original graph. The coproduct is given by disjoint union of the vertex sets with the edges of each graph preserved. The proof of these is trivial.

Equalisers

Given a pair of parallel morphisms $f, g : A \to B$ an equalizer is a universal object E and morphism $e : E \to A$ such that $f \circ e = g \circ e$. Universality is expressed diagrammatically as:



Where D is any object with a morphism, $h : D \to A$, and $\hat{h}$ is the unique morphism that factors through E. A coequalizer, the dual concept, can be obtained by reversing the morphisms.

In **Graph** the equalizer of two graph homomorphisms $f : G_1 \to G_2,\ g : G_1 \to G_2$ is the graph $G_{eq} = (V_{eq}, E_{eq})$ where $V_{eq} = \{v \in V_1 \,|\, f(v) = g(v)\}$ and $E_{eq} = \{e \in E_{eq} \,|\, e \in V_{eq}^2\}$. The homomorphism $e : G_{eq} \to G_1$ is the identity function.

Proof. Any homomorphism $h : G_3 \to G_1$ such that $f \circ h = g \circ h$ must have an image in V_{eq}. Thus, there is also a graph homomorphism $\hat{h} : D \to G_{eq}$. □

Functors

A functor is a map between categories. The action of a functor $F : \mathcal{C} \to \mathcal{D}$ is given by two components:

- An object map: for all $A \in Ob(\mathcal{C})$ there is an object $FA \in Ob(\mathcal{D})$
- A morphism map: for all $f : A \to B$ there is a morphism in $\mathcal{D}$, $Ff : FA \to FB$. This mapping needs to preserve composition and identity, that is, $F(f \circ g) = Ff \circ Fg$ and $Fid_A \to id_{FA}$

A functor from a category to itself is called an endofunctor.

[1]The diagram commutes when all paths from the same start point to the same end point in the diagram give the same result.

Natural Transformations

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\eta : F \to G$ is a family of morphisms in $\mathcal{D}$ such that for every object $X \in Ob(\mathcal{C})$ there is a morphism $\eta_X : F(X) \to G(X)$. This family of morphisms provide a map from the codomain of F to the codomain of G. The morphism must satisfy the following commutative diagram:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ GA & \xrightarrow[Gf]{} & GB \end{array}$$

This condition is called naturality.

2.1.1 Comonads

A comonad over $\mathcal{C}$ is a triple (Q, ϵ, δ) where Q is an endofunctor on $\mathcal{C}$ and $\epsilon : Q \Rightarrow ID_{\mathcal{C}}$, $\delta : Q \Rightarrow Q^2$ are natural transformations satisfying the following commutative diagrams:

$$\begin{array}{ccc} QA & \xrightarrow{\delta_A} & Q^2A \\ {\scriptstyle \delta_A}\downarrow & & \downarrow{\scriptstyle Q\delta_A} \\ Q^2A & \xrightarrow{\delta_{QA}} & Q^3A \end{array} \qquad \begin{array}{ccc} QA & \xrightarrow{\delta_A} & Q^2A \\ {\scriptstyle \delta_A}\downarrow & \searrow & \downarrow{\scriptstyle Q\epsilon_A} \\ Q^2A & \xrightarrow{\epsilon_{QA}} & QA \end{array}$$

$ID_{\mathcal{C}}$ is the identity functor on $\mathcal{C}$. Comonads are the categorical dual of monads, the definition of a monad is the same as above but with all morphisms reversed.

A comonad can also be given in coKleisli form. For this you need:

- An object map: Q
- For every object $A \in Ob(\mathcal{C})$ a counit morphism: $\epsilon_A : QA \to A$
- A Kleisli coextention operation $(*)$ which takes $f : QA \to B$ to $f^* : QA \to QB$.

The coKleisli form of the comonad must satisfy the following equations:

$$\epsilon_A^* = id_{QA} \qquad \epsilon \circ f^* = f \qquad (g \circ f^*)^* = g^* \circ f^*$$

The coKleisli and standard form of the comonad are equivalent. To go from coKleisli to standard form we:

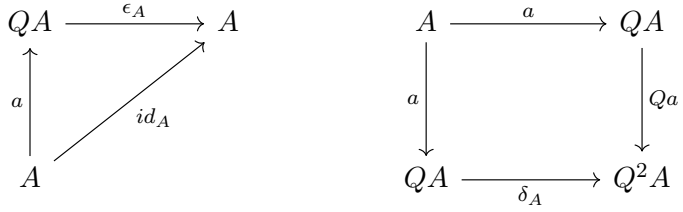
- extend G to a functor by $Qf = (f \circ \varepsilon)^*$
- define δ_a as id^*_{QA}

Then we have a comonad (ϵ is the same for both). To go the other way, we can define Kleisli coextention as $f^* = Qf \circ \delta_A$ and take action on objects from the functor Q.

Using this definition of a comonad we can define a coKleisli category $Kl(Q)$. The objects of the category as the objects of $\mathcal{C}$ and morphisms from A to B are given by the morphisms in $\mathcal{C}$ of the form $QA \to B$. Composition in the coKleisli category, $\bullet$, is given by composition of is given by $g \bullet f := g^* \circ f$ for $f : QA \to B$ with $g : QB \to C$.

Coalgebras

A coalgebra over a comonad (Q, ϵ, δ) on category $\mathcal{C}$ is a pair (A, a) where $A \in \mathcal{C}$ and $a : A \to QA$. It satisfies the following coherence conditions:



2.2 Finite Model Theory

Finite model theory is the study of logics over finite models. It has many applications in computer science, from database theory to computational complexity [16]. We start by defining sentences over models for a given logic followed by models themselves.

For a particular logic, sentences are built from the standard logic symbols $(\forall, \exists, \wedge, \vee, \neg)$, variables, and predicates that are specified by a relational signature[2] σ, a set of symbols $R \in \sigma$ each associated with a strictly positive integer arity n.

A model, or σ-structure, $\mathscr{A}$, is an interpretation of a relation signature. It consists of a set A, called the universe, along with a set $R^{\mathscr{A}} \subseteq A^n$ for each R of arity n in σ. The tuples in these sets are denoted $R^{\mathscr{A}}(a_1, \ldots, a_n)$.

We can define homomorphisms between these models. Given σ-structures $\mathscr{A}$ and $\mathscr{B}$, a homomorphism $h : \mathscr{A} \to \mathscr{B}$ is a function $h : A \to B$, such that, for each relation symbol R of arity n in σ, for all $a_1, \ldots, a_n \in A$: $R^{\mathscr{A}}(a_1, \ldots, a_n) \Rightarrow R^{\mathscr{B}}(h(a_1), \ldots, h(a_n))$. The reader may notice this is a generalization of the graph homomorphisms defined at the start of section 2.1.

In fact, we can define a category for each of these relational signatures σ. The objects are σ-structures and the morphisms are the homomorphisms between them. This category is denoted $\mathcal{R}(\sigma)$.

It so happens that if we take the signature σ_G consisting of a single symbol $\frown$ of arity 2 then the category $\mathcal{R}(\sigma_G)$ is isomorphic to **Graph**.

Proof. A model $\mathscr{A} \in Ob(\mathcal{R}(\sigma_G))$ with a universe A is a graph $G = (A, \frown^{\mathscr{A}})$. Similarly, a graph $G = (V, E)$ corresponds to a model $\mathscr{A}$ with a universe V and $\frown^{\mathscr{A}} = E$. The definition of a graph morphism exactly matches the definition of a homomorphism in this category. □

2.2.1 Logics

In this paper we will be looking at fragments and extensions of the infinitary logic $\mathcal{L}_{\infty,\omega}$, for brevity we denote this simply $\mathcal{L}$. This logic is an extension of first order logic with infinitary conjunctions and disjunctions, but where formulas only contain a finite number of variables. Fragments of interest are

- $\mathcal{L}^k$ – only formulas of $\mathcal{L}$ with quantifier rank $\leq k$

[2] Also called a relational vocabulary

- $\mathcal{L}_k$ – only formulas of $\mathcal{L}$ with k variables or less
- $\exists\mathcal{L}$ – the existential positive fragment of $\mathcal{L}$, formulas only made up of conjunction, disjunction and existential quantifiers.
- $\mathcal{L}(\#)$ – the extension of $\mathcal{L}$ with counting quantifiers. These are of the form $\exists_{\leq n}$, $\exists_{\geq n}$, where the semantics of $\mathscr{A} \models \exists_{\geq n} x.\, \psi$ is that there exist at least n distinct elements of A satisfying ψ.

2.2.2 Model comparison games

In model theory we are often interested in equivalence between σ structures under particular logics. Model comparison games are a way of proving this [8, 10, 15].

The games are played over a pair of models $\mathscr{A}$ and $\mathscr{B}$, between two players, the Spoiler and the Duplicator. The Spoiler wants to show the models are different while the Duplicator wants to prove they are the same.

The exact details of how the game is played differ from game to game, they are all generally bounded by some resource parameter k.

Ehrenfeucht-Fraïssé Games

The Ehrenfeucht-Fraïssé game played by the Spoiler and the Duplicator. The game has k rounds an is played over two structures $\mathscr{A}$ and $\mathscr{B}$. On the i'th round, $1 \leq i \leq k$:

- The Spoiler chooses an element from one of the structures, either $a_i \in A$ or $b_i \in B$
- The Duplicator then picks an element from the other structure, either $b_i \in B$ or $a_i \in A$

The Duplicator wins the k round game if at the end of the game the relation $r := \{(a_i, b_i) \mid 1 \leq i \leq k\}$ is a partial isomorphism.

The game is won by the Duplicator if at the end they have a *partial isomorphism*. A partial isomorphism between two models $\mathscr{A}$ and $\mathscr{B}$ with signature σ is a relation $r \subseteq A \times B$ such that:

- For all $(a_i, b_i), (a_j, b_j) \in r$ we have $a_i = a_j$ iff $b_i = b_j$.
- For every n-ary relation $R \in \sigma$ for any $R^{\mathscr{A}}(a_1, \ldots, a_n)$ where $(a_1, b_1), \ldots, (a_n, b_n) \in r$ then we must have $R^{\mathscr{B}}(b_1, \ldots, b_n)$.

There is also an asymmetric variant of the Ehrenfeucht-Fraïssé game where Spoiler can play in either only $\mathscr{A}$ and the Duplicator in only $\mathscr{B}$.

Pebble games

The pebble game is similar to the Ehrenfeucht-Fraïssé game except here the parameter k represents the number of available pebbles. These pebbles are number 1 to k. Unlike Ehrenfeucht-Fraïssé games, there is no limit on the number of rounds. In the k pebble game over structures $\mathscr{A}$ and $\mathscr{B}$ on the j'th round, $1 \leq j$:

- The Spoiler places one of his pebbles p_i, $1 \leq i \leq k$, on some element from one of the structures, either $a \in A$ or $b \in B$. If p_i was previously placed on some other element of the structure then he removes it and replaces it on a.

- The Duplicator then picks an element from the other structure and places her corresponding pebble p_i, either on $b \in B$ or $a \in A$. Again she may have to move it from somewhere else.

At round n of the game we can form a relation r from the latest placing of each pebble p_i. Duplicator wins the play on round n if r forms a partial isomorphism. Duplicator only wins the k-pebble game if she has a strategy which wins for all n.

Logical Equivalence

Due to well established results these games characterize logical equivalence as follows:

Theorem 1 ([8], [15])**.** *Let $\mathscr{A}$ and $\mathscr{B}$ be σ-structures.*

- *Duplicator has a winning strategy in the k-round Ehrenfeucht-Fraïssé game from $\mathscr{A}$ to $\mathscr{B}$ iff $\mathscr{A}$ and $\mathscr{B}$ satisfy the same sentences of quantifier rank $\leq k$.*
- *Duplicator has a winning strategy in the k-pebble game from $\mathscr{A}$ to $\mathscr{B}$ iff $\mathscr{A}$ and $\mathscr{B}$ satisfy the same sentences of k-variable logic.*

2.2.3 Gaifman Graphs

A useful concept to define on a σ-structures is the Gaifman Graph.

Given a σ-structure $\mathscr{A}$, the Gaifman graph $\mathcal{G}(\mathscr{A}) = (A, E)$ is a graph with vertices A where $(a, a') \in E$ iff for some relation $R \in \sigma$, for some $(a_1, \ldots, a_n) \in R^{\mathscr{A}}$, $a = a_i$, $a' = a_j$, $a \neq a'$. When σ is the signature for a graph (contains only a single binary relation) the Gaifman graph is the undirected irreflexive version of this graph. The concept allows us to define properties of graphs over relational structures.

Chapter 3

Comonads

In this chapter we give the formal definitions, in coKleisli form, of the comonads that this paper implements in Haskell.

3.1 The Ehrenfeucht-Fraïssé Comonad

The Ehrenfeucht-Fraïssé comonad provides a way to capture asymmetric Ehrenfeucht-Fraïssé games over two structures $\mathscr{A}$ and $\mathscr{B}$ [2].

The Ehrenfeucht-Fraïssé comonad $\mathbb{E}_k$ is an indexed comonad over $\mathcal{R}(\sigma)$ meaning that a separate comonad is defined for every positive integer k. To define each comonad, we give the object map $\mathbb{E}_k$, the counit $\varepsilon_{\mathscr{A}}$, and the coextension $(\_)^*$.

We start with the object map. Given a structure, $\mathscr{A}$, let $\mathbb{E}_k\mathscr{A}$ be the σ-structure with a universe of non-empty lists of length at most k made up of elements of A, that is, $\mathbb{E}_k A := A^{\leq k}$. Intuitively these are plays in the Ehrenfeucht-Fraïssé game of length at most k.

To define the relations $R^{\mathbb{E}_k\mathscr{A}}$ on $\mathbb{E}_k\mathscr{A}$ we first define the counit. The counit $\varepsilon_{\mathscr{A}} : \mathbb{E}_k A \to A$ is the function $\varepsilon_{\mathscr{A}}[a_1, \ldots, a_j] = a_j$. For a n-ary relation R, $R^{\mathbb{E}_k\mathscr{A}}$ is defined as the set of length n tuples $(s_1, \ldots, s_n)$ of lists which are pairwise comparable in the prefix ordering, that is for any two lists s_i, s_j one must be a prefix of the other. This captures the idea that two lists are only related if they are both part of the same play in the Ehrenfeucht-Fraïssé game. Also, the tuple must satisfy $R^{\mathscr{A}}(\varepsilon_{\mathscr{A}} s_1, \ldots, \varepsilon_{\mathscr{A}} s_n)$ *i.e.* if we take the last element of each sequence we get a tuple in $R^{\mathscr{A}}$.

Finally, we define the coextension. Given a homomorphism $f : \mathbb{E}_k\mathscr{A} \to \mathscr{B}$, we define $f^* : A^{\leq k} \to B^{\leq k}$ by $f^*[a_1, \ldots, a_j] = [b_1, \ldots, b_j]$, where $b_i = f[a_1, \ldots, a_i]$, $1 \leq i \leq j$.

$\mathbb{E}_k\mathscr{A}$ can be thought of as a structure containing all possible Spoiler strategies. The Duplicators strategies are then represented by a homomorphism $h : \mathbb{E}_k\mathscr{A} \to \mathscr{B}$. h takes a play $s = [a_1, \ldots, a_i]$ to the Duplicator's response to a_i, *i.e.* b_i. Since $\mathbb{E}_k\mathscr{A}$ contains all possible Spoiler plays (including all prefixes of s) this gives a valid strategy. The full proof of the equivalence between $h : \mathbb{E}_k\mathscr{A} \to \mathscr{B}$ and a Duplicator strategy is given in [2].

3.1.1 Ehrenfeucht-Fraïssé Coalgebras

The coalgebras of the Ehrenfeucht-Fraïssé comonad are homomorphisms $\mathscr{A} \to \mathbb{E}_k\mathscr{A}$ that satisfy the coalgebra conditions given in section 2.1.1. It turns out that there is a bijective correspondence between a coalgebra $\mathscr{A} \to \mathbb{E}_k\mathscr{A}$ and tree depth decomposition of depth k of the structure $\mathscr{A}$ [2].

The tree depth of a structure $\mathscr{A}$ is given by the tree depth of its Gaifman graph (see section 2.2.3). This is defined through the *tree depth decomposition*.

A tree depth decomposition of a graph G is a forest composed of trees T such that for each tree every edge of G connects a pair of nodes that have an ancestor-descendant relationship in some T. Each tree corresponds to a connected component of G.

The formal proof of this isomorphism can be found in [2]. The coalgebra equations equates to saying the function must be an injective map to a prefix closed subset of $A^{\leq k}$. This, combined with the homomorphism condition, gives us that the coalgebra maps the vertices of $\mathcal{G}(\mathscr{A})$ to its tree depth decomposition in $\mathbb{E}_k\mathscr{A}$.

3.2 Pebbling Comonad

The pebbling comonads, $\mathbb{P}_k$, are similar but different to $\mathbb{E}_k$. The object map takes a structure $\mathscr{A}$ to $\mathbb{P}_k\mathscr{A}$. A turn in the pebble game consists of placing a pebble p_i on some element of $\mathscr{A}$, $a \in A$. Thus the universe of $\mathbb{P}_k\mathscr{A}$ is $(\mathbf{k} \times A)^+$ where $\mathbf{k}$ is the set of integers from 1 to k. This is the set of non-empty lists containing pairs of a pebble index and an element $a \in A$. Note that unlike the in the universe of $\mathbb{E}_k\mathscr{A}$ the lists here can be of any length.

The counit $\varepsilon_{\mathscr{A}} : \mathbb{P}_k\mathscr{A} \to \mathscr{A}$ is defined as $\varepsilon_{\mathscr{A}}[(p_1, a_1), \ldots, (p_n, a_n)] = a_n$. For an n-ary relation $R \in \sigma$, we have $R^{\mathbb{P}_k\mathscr{A}}(s_1, \ldots, s_n)$ iff the following conditions hold

- All s_i are pairwise comparable in the prefix ordering, *i.e.* for any two s_i, s_j either s_i is a prefix of s_j or vice versa.
- The pebble index of the last move in each s_i does not appear in the suffix of s_i in s_j where s_i is a prefix of s_j, for any s_j.
- We have $R^{\mathscr{A}}(\varepsilon_{\mathscr{A}}(s_1), \ldots, \varepsilon_{\mathscr{A}}(s_n))$ in $\mathscr{A}$.

Here the second condition enforces that only the current position of a pebble is taken into account when deciding whether elements are related.

Finally, given a homomorphism $f : \mathbb{P}_k\mathscr{A} \to \mathscr{B}$, we define $f^* : \mathbb{P}_k\mathscr{A} \to \mathbb{P}_k\mathscr{B}$ by $f^*[(p_1, a_1), \ldots, (p_j, a_j)] = [(p_1, b_1), \ldots, (p_j, b_j)]$, where $b_i = f[(p_1, a_1), \ldots, (p_i, a_i)]$, $1 \leq i \leq j$.

Again $\mathbb{P}_k\mathscr{A}$ contains all possible Spoiler strategies and Duplicator strategies are represented by homomorphisms $h : \mathbb{E}_k\mathscr{A} \to \mathscr{B}$. The full proof of this equivalence is in [1].

The coalgebras of the pebbling comonad, like the Ehrenfeucht-Fraïssé comonad, also correspond to an important combinatorial parameter of the σ-structures, the tree width [2]. Since these are not covered in the implementation we refrain from giving a full description here.

3.2.1 Logical equivalence

Both the comonads above give rise to characterizations of logical equivalence. The Ehrenfeucht-Fraïssé comonad, like Ehrenfeucht-Fraïssé games, is associated with the logic $\mathcal{L}^k$ (as defined in section 2.2.1) and the pebbling is associated with the logic $\mathcal{L}_k$. In [2] it is shown that if there are two morphisms $\mathbb{E}_k\mathscr{A} \to \mathscr{B}$ and $\mathbb{E}_k\mathscr{B} \to \mathscr{A}$ then $\mathscr{A} \equiv^{\exists\mathcal{L}^k} \mathscr{B}$. Similarly for the pebbling comonad, if these two coKleisli morphisms exist then we have $\mathscr{A} \equiv^{\exists\mathcal{L}_k} \mathscr{B}$.

On top of this, if the two morphisms form an isomorphism, *i.e.* $\mathbb{E}_k\mathscr{A} \to \mathscr{B}$ is the inverse of $\mathbb{E}_k\mathscr{B} \to \mathscr{A}$, then we have $\mathscr{A} \equiv^{\mathcal{L}^k(\#)} \mathscr{B}$. Similarly, for the pebbling comonad, we would have $\mathscr{A} \equiv^{\mathcal{L}_k(\#)} \mathscr{B}$.

Spans

The reader may notice we do not capture equivalence in the pure logics $\mathcal{L}_k$, $\mathcal{L}^k$, instead we can only prove equivalence a fragment or an extension of them. Capturing the logic exactly is not easy. In [2] a method is presented to capture this back and forth equivalence for all the comonads. The method involves the use of the categorical concept of spans. Spans can be though of as a generalization of morphisms where there is no longer any asymmetry between source and target. Implementing spans in Haskell was not in the scope of this paper but would make a valuable extension to the work.

Chapter 4

Design Decisions

In this chapter we discuss the decisions made when implementing the comonads and how they have affected the final outcome. Alongside this we introduce the Haskell extensions used in chapter 6.

4.1 Why Haskell

Haskell is a widely used programming language. It is well known for its categorical concepts and has many libraries relevant to the project. It is also a language the author is familiar with. Haskell is generally well suited to the task of implementing the game comonads, the largest obstacle however is its lack of *dependent types*.

Dependent types allow for function types to be derived from concrete values. Haskell has some support for this though it is mostly unofficial and has a steep learning curve. In our case it would be a lot easier to parameterize the category of relational structures by a signature σ using dependent types. It is the hoped that someday Haskell will have the full functionality of dependent types. For these reasons we have chosen to do this project in Haskell rather than a less well-used language such as Idris or Agda.

4.1.1 Idris and Agda

Both Idris and Agda are purely functional language that implement dependent types. Both have similar syntax to Haskell but unlike Haskell they are both designed to be used as proof assistants. Idris has category theory libraries [12] and is a good candidate to continue the work in this paper, however, the current environment and library support is not as mature as Haskell. Similarly, category theoretic work exists in Agda [13] but it is much more recent and underdeveloped than that of Haskell.

Idris is designed to be more of a general purpose programming language than Agda, which is more focused proofs. Idris also shares more syntax with Haskell, however Agda code can be compiled to Haskell. Both of these languages are good candidates in which to extend the work of this paper.

4.2 Restricting to graphs

Without dependent types implementing parameterized categories in Haskell becomes complex very quickly. It is for this reason that we decided to focus only on a single relational signature. It is often said about finite model theory that all the interesting things can be done with graphs. For this reason we choose the category $\mathcal{R}(\sigma_G)$ where the signature

contains a single predicate $\frown$ of arity 2. As discussed in section 2.2, this category is isomorphic to the category **Graph**.

4.2.1 Algebraic Graphs

The Haskell library `algebraic-graphs` provides a type safe implementation of graphs in Haskell [17]. Type safe means that, unlike many other implementations of graphs in Haskell, every member of the type is a valid graph. For example representing a graph as the type `data Graph a = G { vertices::[a], edges::[(a,a)]}` allows graphs with an edge over a value not in the list of vertices. Other major graph libraries suffer from similar problems [17]. Algebraic graphs fix this by providing a simple class interface for directed graphs that only allows for correct graphs to be built. The algebraic `Graph` class is:

```
class Graph g where
        type Vertex g
        empty   :: g
        vertex  :: Vertex g -> g
        overlay :: g -> g -> g
        connect :: g -> g -> g
```

The function `empty` builds the empty graph and `vertex` creates a graph with a single vertex. The functions `overlay` and `connect` can be used to merge graphs. `overlay` places two graphs next to each other merging vertices of the same value, formally:

$$overlay\ (V_1, E_1)\ (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$$

`connect` connects every edge in the first graph to every edge in the second graph, formally:

$$connect\ (V_1, E_1)\ (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$$

Members of the `Graph` class will be objects in the Haskell implementation of the category $\mathcal{R}(\sigma_G)$.

4.3 Categories In Haskell

Some argue that the whole of Haskell can be viewed as a category **Hask** where functions are morphisms and objects are types[1] [11]. Haskell even includes definitions for endofunctors and monads over **Hask** as part of the standard Prelude:

```
class Functor f where
        fmap :: (a -> b) -> f a -> f b

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

This is all well and good if the category you are interested in is **Hask**, however, we are instead interested in $\mathcal{R}(\sigma_G)$. Luckily, for any σ, $\mathcal{R}(\sigma)$, can be viewed as a subcategory of **Hask** and there exists some infrastructure for building these [18].

[1] While this is a valid category it does not have the nice properties expected, such as `Either` being sum and members of the monad class `Monad` being a monad (provided they satisfy the laws). The issue is that the bottom value $\perp$ allows for multiple different functions in cases where one might hope for a single unique function. To remedy this we generally think of **Hask** as excluding $\perp$, in this version Haskell Monads are in fact monads and all the properties we expect fall out. Though some still argue this is not a valid category [3]

4.3.1 Approaches to subcategories of Hask

This section outlines existing libraries and ideas that can be used to work with categories in Haskell. We do not use an existing approach, rather, we combine the best aspects of many. The language extensions explained in this chapter are heavily used in 6 (e.g. GADTs, Constrains, Kinds).

Existing approaches

The most obvious starting place for custom categories in Haskell is the `Control.Category` package along with its children. This package defines a category class that specifies composition and identity of some `cat` class which represents the morphisms. Unfortunately there is no way to constrain the objects of the category, only the morphisms. In any category implemented using this all types are valid objects.

There are two major approaches to the specifying the objects of the category. The most common approach is using constraint kinds to limit the types of the objects, the other approach is using generalized algebraic data types (GADTs).

Categories with Generalised Algebraic Data Types

GADTs are an offshoot from dependent type theory that among other things, allows us to construct data types with a restricted domain. For example:

```
data T a b where
        T1 :: => [(a,b)]    -> T a b
        T2 :: => Either a b -> T a b
```

Here we have used a GADTs `data` definition to specify a type `T`. The domain constructor `T1` is restricted to lists of pairs and similarly `T2` can only be used on values of the type `Either a b`. This means that when writing a function over `T` we can pattern match against the constructors and get guarantees about what the `x` in `T1 x` is (i.e. that `x` is a list of pairs).

In the `data-category` library, GADTs are used to restrict the types of morphisms. This library however, only represented objects by their identity arrows. This makes it very hard to reason about, manipulate and store the objects themselves *i.e.* graphs. It is also hard express subcategories. With constraints and classes this is easy (due to inheritance).

Categories with Constraints

Using class constraints to restrict the objects and morphisms of a category is the most common approach to building subcategories of **Hask** in Haskell.

Constraints in Haskell are classes that can be used to restrict arguments of a function to only members of the given class. For example, a function of type `Eq a => a -> a` only allows types that have equality implemented over them. The compiler extension `ConstraintKinds` allows for more flexibility in what constitutes a constraint.

The `ConstraintKinds` extension works by allowing more members of the `Constraint` *kind*. Kinds can be though of as higher order types, *i.e.* types of types. Regular types all have the kind `*`, data constructors can have more complex kinds, for example:

```
Int :: *
Maybe :: * -> *
Maybe Bool :: *
Either :: * -> * -> *
```

There is also a kind called `Constraint`. Classes have kinds that end in `Constraint` *e.g.* :

```
Eq :: * -> Constraint
Show :: * -> Constraint
Functor :: (* -> *) -> Constraint
```

The `ConstraintKinds` extension allows us to use the `Constraint` kind as a first class citizen, enabling us to parameterize by constraints. In particular, we can now give a definition for a functor that restricts the domain to only types that satisfy some constraint:

```
class RFunctor f where
        type SubCat f a :: Constraint
        rfmap :: (SubCat f a, SubCat f b) => (a -> b) -> f a -> f b
```

We can use this `SubCat` constraint to specify the objects of our category, *i.e.* only types that implement the class specified by `SubCat`.

Libraries that use this approach were initially considered for this project, however, in the end all were found in some way wanting.

- `constrained-categories` – This is a large and powerful library but was rather overcomplicated for our use case. The functor implementation also had a strange dependency on the category being monoidal (it used the monoid unit object in the definition) which, while it would be possible to endow the category of graphs with monoidal structure, was not particularly necessary.

- `subhask` – Another large library too elaborate for our needs. The library completely rewrites the prelude and introduces subcategories of Haskell on the side. It is also optimized for vector space categories, using the `hmatrix` package in the category definitions.

- `hask` – A well-made library that is faithful to the category theoretic concepts however is more optimized for programming use than theoretical category implementation. It is designed for uses with lenses.

The eventual decision was to take the best parts from all the libraries and implement our own categories in Haskell. We used a mixture of Constraint and GADTs.

There was one last major compiler extension used that bears mention here, `KindSignatures`. The extension `KindSignatures` allows us to restrict the kind of class parameters *e.g.* `class C (f :: * -> *) a where ...` restricts the kind of the type `f` to be `*->*` *i.e.* a class with one parameter.

We have now covered all the compiler extensions used to build the subcategories and now move on to the details of the implementation.

Chapter 5

Overview/Interface

In this chapter we give an overview of the primary features of the implementation as well as listing the relevant functions and classes. Everything mentioned in this chapter is explored in depth in chapters 6 and 7. There are also may more functions not mentioned here that are discussed in chapter 6.

5.1 Features

We now list the main features of the implementation followed by the functions, classes and class instances that implement them:

- An implementation of subcategories of **Hask**. **Hask** is the category with Haskell types as objects and Haskell functions as morphisms. The implementation uses constraints to restrict the types in the category. There are functor, bifunctor and comonad classes over these categories as well as coKleisli categories for each comonad instance.
  - `Category` class
  - `CComonad` class
  - `CFunctor` class
  - `CBifunctor` class
  - `CoKleisli` instance of `Category`
- An implementation of the category of graphs. The objects of the category are any type implementing the `Graph` class from the `algebraic-graphs` library. The morphisms are vertex maps. These are a subset of graph homomorphisms.
  - `GraphMorphism` instance of `Category`
- Instances of the indexed families of Ehrenfeucht-Fraïssé and Pebbling comonads, $\mathbb{E}_k$ and $\mathbb{P}_k$, over the category of graphs. There are also products and coproducts over the category.
  - `EF` instance of `CComonad`
  - `Peb` instance of `CComonad`
  - `Product` instance of `CBifunctor`
  - `Coproduct` instance of `CBifunctor`

- Functions to check equivalence in the logics $\exists\mathcal{L}^k$ and $\mathcal{L}^k(\#)$ by checking for the appropriate coKleisli morphisms. Also functions to check the validity of homomorphisms, $\mathbb{E}_k$ graphs, and $\mathbb{P}_k$ graphs. To check the equivalence we need a way to generate the graph $\mathbb{E}_k\mathscr{A}$ for any graph $\mathscr{A}$ so we have a function `graphToEFk` which generates this.
  - `eqQRankKEPfrag`
  - `eqQRankKCounting`
  - `checkMorphIsHomo`
  - `checkValidEFkGraph`
  - `checkValidPebkGraph`
  - `graphToEFk`
- Functions that convert $\mathbb{E}_k$ coalgebras to tree depth decompositions and back as well as a function to get the Gaifman graph of a graph. There are also functions to check the validity of coalgebras and tree depth decompositions
  - `forestToCoalg`
  - `coalgToForest`
  - `getGaifmanGraph`
  - `checkValidCoalg`
  - `checkValidTDD`

5.2 Examples

Here we present a list of the examples used in chapter 7 to illustrate the features above.

- An example that proves the theorem: for two linear orders of length at least 2^k we have $L_1 \equiv_{\exists\mathcal{L}^k} L_2$.
- Proof that evenness, 2-colourability and connectivity are not expressible in $\exists\mathcal{L}^k$
- A function that proves Hamiltonicity is not expressible in first order logic.

Chapter 6

Implementation

In this chapter we explain in depth each component of the implementation. For brevity, we have omitted many of the full definitions and instead focused on communicating the ideas behind them. The full implementation can be found in the appendix.

6.1 Categories

As discussed in section 4.3.1 none of the existing category theory libraries for Haskell were quite suited to our use case. Hence, we defined our own custom categories. The main ingredients used in our implementation are Constraints and GADTs (see section 4.3.1). Other, less instrumental compiler extensions were also used. For a full list please refer to the code.

6.1.1 Category framework

We must naturally start with our Haskell definition of a category. To define a category we must specify four things:

- The objects
- The morphisms
- The *id* morphism for every object
- How to compose morphisms

We use a constraint to give the subset of types that make up the objects of the category. We then use a data type (or GADT) with two type parameters for the morphisms (the two parameters being the source and target object). The `id` and composition function `(.)` can then be defined in terms of these data morphisms. The morphism type is used to identify the category. The Haskell definition is:

```
class Category (cat :: k -> k -> *) where
    type Object cat (o :: k) :: Constraint
    id :: (Object cat o) => cat o o
    (.) :: (Object cat a, Object cat b, Object cat c) =>
    cat b c -> cat a b -> cat a c
```

Unfortunately the use of the constraints does not lead to particularly transparent code. The mix of compiler extensions here can take some getting used to. The `cat` data constructor that characterizes the category is of kind `k -> k -> *` where `k` is the kind of the

objects of the category. This data type represents the morphisms of the category, hence why it takes two objects and returns a type, *i.e.* the type of the morphisms.

The constraint `Object cat a` may appear strange in that it has two variables `cat` and `a`, but this has a rational explanation. When we declare an instance of a category we specify something like `type Object MyCat g = Eq g`. So the `Object cat` part of the constraint would in this case reduce to `Eq`.

On the first line of the class we specify that the type `Object cat (o :: k)` is of kind `Constraint`. The `(o :: k)` part specifies that the thing being constrained is of the same kind as the objects the morphisms act on, fixed by `cat :: k -> k -> *`.

By separately specifying the objects and morphisms we allow for the interactions between them to be specified elsewhere, (for instance how to use a morphism and a value from its source type to map to a value in the target type).

Functor and Comonad classes

Now that we have a category class we can go on to define other concepts on top of it.

The definition for a relative functor (`CFunctor`) looks similar to a regular Haskell `Functor` except with constraints dotted around to ensure it has the correct domain an codomain categories. See the code for the full definition of `CFunctors` and `CBifunctors`.

Now on to the main event: the definition for a relative comonad. To understand the definition it is clearer to examine a **Hask** comonad rather than going straight to the comonad, `CComonad`, over arbitrary categories. The definition given here, like `CComonad`, is in coKleisli form, though the regular for is also automatically generated[1].

```
class Comonad w where
    counit :: w a -> a
    extend :: (w a -> b) -> w a -> w b
    duplicate :: w a -> w (w a)
    duplicate = extend id

instance (Comonad f) => Functor f where
    fmap f = extend (f . counit)
```

The definition for the relative comonad `CComonad` used in the code is effectively the same with numerous extra constraints on the objects and has custom morphisms instead of Haskell functions. We refer readers to the code for the full definition.

CoKleisli categories

Every comonad automatically induces a coKleisli category. We can give the following definition which automatically defines the coKleisli category over every `CComonad` instance:

```
data CoKleisli f cat a b where
    CoKleisli :: (Category cat, CComonad f cat) =>
        cat (f a) b -> CoKleisli f cat a b

instance Category (CoKleisli f cat) where
```

[1]Note that this does not work the other way around. Defining `counit` and `duplicate` will not give `extend` since `Comonad` is not necessary a functor, just an object map (it would have to be a subclass of `Functor` to get a definition for `extend`).

```
type Object (CoKleisli f cat) a =
    (Category cat, CComonad f cat,
     Object cat a, Object cat (f a))
id = CoKleisli counit
(CoKleisli f) . (CoKleisli g) = CoKleisli (f . extend g)
```

We can now refer to the coKleisli morphisms for any comonad.

6.2 Game Comonads

We now come on to the implementation of the game comonads themselves. To define a `CComonad` we first need to specify the category it acts over.

6.2.1 Category of Graphs

Ideally, this would be the category of relational structures but as discussed in section 4.2 for this implementation, the category of graphs $\mathcal{R}(\sigma_G)$ is used. As discussed in section 4.2.1 the Algebraic Graphs library provides a class, `Graph`, that captures types that implement algebraic graphs. This class can be used as a constraint to restrict the objects of the category. We assume, as a precondition, that any `Graph` will merge vertices with the same value.

Theoretical grounding

Each type `g` implementing `Graph` has an underlying vertex type `Vertex g`. For a σ_G-structure $\mathscr{A}$, we think of the universe A as containing precisely the elements of the type `Vertex g`.

Representing a morphism $h : \mathscr{A} \to \mathscr{B}$ is not so simple. We want to use the type to represent only valid homomorphisms. One approach is to represent them as a pair of graphs and a mapping between them. Haskell however, does not have any way to add constraints like the condition for a valid homomorphism at type level.

Instead, we have chosen to represent them as vertex maps. The advantage of this is that it allows for a very simple definition for a graph morphism.

```
data GraphMorphism a b where GM :: (Graph a, Graph b)
    => (Vertex a -> Vertex b) -> GraphMorphism a b
```

A graph morphism can be used to transform a graph of type `a` to a graph of type `b` by applying the vertex map. This is necessarily a subset of valid homomorphisms. Due to the merging of vertices of the same value, the number of edges of the target graph will be less than or equal to that of the source graph.

Any member of a type `GraphMorphism a b` contains a function of type `f :: Vertex a -> Vertex b`. Since the objects of this category are graphs, not types, this function actually represents multiple graph homomorphisms. By apply the function to a specific graph we effectively pick out the graph homomorphism that goes from `g -> gmap f g` where `g` is a graph and `gmap` is the vertex map function.

On top of this we consider a graph with vertex type `Vertex a` to have a vertex at every value in the type. Some of these vertices are in edges of the graph some are not. When we apply `gmap f`, it only applies the function `f` to vertices in edges. We can

therefore view a graph homomorphism `g -> gmap f g` instead as yet another family of graph homomorphisms that contain every possible map on vertices not in any edge.

In reality these details don't make much difference to the structure of the implementation, only the true validity of the proofs.

We declare the category of graphs as an instance of `Category` as follows:

```
instance Category GraphMorphism where
    type Object GraphMorphism g = Graph g
    id          = GM Prelude.id
    GM f . GM g = GM (f Prelude.. g)
```

Practical problems

There is one small issue we must address. The `Graph` class contains only functions for constructing graphs; there is no way to perform actions such as a vertex map to get the edges of the graph. To get round this issue we can specify a new class, `GraphCoerce`, and a base representation, the `AdjacencyMap`. The class specifies function to transform any of type implementing `GraphCoerce` into an `AdjacencyMap` and back. `AdjacencyMaps` are a graph representation from the `algebraic-graphs` library which have a wide selection of functions for manipulating and deconstructing the graphs including `gmap` and `edges` which returns a list of the edges in the graph as pairs.

```
class Graph g => GraphCoerce g where
    gcoerce    :: g -> AdjacencyMap (Vertex g)
    gcoerceRev :: AdjacencyMap (Vertex g) -> g
```

To build functors over the category we can add GADTs data constructors that take a `Graph` of the appropriate form. For reasons outlined at the start of the next section in the current implementation the functors only take `AdjacencyMaps`, however, with the use of the `GraphCoerce` class any graph can still have a functor applied to it. For each functor data type we need to make it a member of the graph class and define `GraphCoerce` over it.

Arguably the objects in the category are now only members of the `GraphCoerce` class, however this makes little practical difference.

6.2.2 Ehrenfeucht-Fraïssé comonad

The Ehrenfeucht-Fraïssé comonad as defined in [2] is an indexed family of comonads defined at every positive integer k. The implementation we give here only defines a single `CComonad` instance in Haskell. This comonad can be considered as the whole family of comonads, with a different invariant for every member of the family.

To define the comonad we first need to give the object map, *i.e.* the data type that can be applied to objects *i.e.* instances of `Graph`. The definition is used in the implementation is:

```
data EF a where EFdata :: (Graph a) =>
    AdjacencyMap [Vertex a] -> EF a
```

As can be seen, to apply the Ehrenfeucht-Fraïssé object map we need to give it an `AdjacencyMap`. This is not a strictly correct definition of an object map. The objects of the category are any types that implement the `Graph` class, here we are only accepting `AdjacencyMap` graphs where the vertices are lists with the base type of the lifted graph.

To apply the functor to any graph we need to have `GraphCoerce` defined on it and use `gcoerce` to turn it into an `AdjacencyMap`. It may be possible to implement the functor with the type signature:

```
data EF a where EFdata ::
    (Graph a, Graph b, Vertex b ~ [Vertex a], GraphCoerce b) =>
    b -> EF a
```

The constraint `Vertex b ~ [Vertex a]` says the two types must be equal. This definition would allow the data constructor to be applied to any type with `GraphCoerce` implemented over it. Sadly the current state of Haskell's dependent type workarounds makes implementing the functor with this definition much more complicated than it should be. Since the type `b` is not known at compile time, most functions over the EF graphs refuse to compile. In the end it was decided that the implementation would be more usable and extensible with the current approach.

Now we have the Ehrenfeucht-Fraïssé comonad object map defined we need to give the comonad over it. The actions of the counit and extend function both turn out to translate to basic Haskell functions making the definition fairly straight forward:

```
instance CComonad EF GraphMorphism where
    counit        = GM last
    extend (GM f) = GM (map f . ninits)

ninits :: [a] -> [[a]]
ninits [] = []
ninits xs = ninits (init xs) ++ [xs]
```

The function `ninits` is `inits` for non-empty lists. To complete the definition we also need to add the invariant for every k:

$$\exists k \in \mathbb{Z}^{+}.\ \forall\ \texttt{Graph A},\ \texttt{xs} \in (\texttt{universe}\ (\texttt{EF A})).\ 1 \leq \texttt{length xs} \leq k$$

Ideally we would restrict the size of the list at the type level and have a parameterized comonad. This is possible with certain extensions in Haskell[2] however it has not been implemented due to time constraints.

To verify that this is in fact a valid comonad we need to prove the three laws for a comonad in coKleisli form. The Haskell versions of these are:

1. `extend counit = id`
2. `counit . extend f = f`
3. `extend f . extend g = extend (f . extend g)`

Proof of comonadicity

1: `extend counit = id`

Proof.

[2]It is possible to implement type level bounded length lists in Haskell, see the package `GHC.TypeNats` for more details

```
    extend counit
    -- defn. of extend
  = GM map last . ninits
    -- Lemma 1
  = GM id
```

Lemma 1: `map f . ninits = id`
Base case:

```
    (map last . ninits) [x]
  = map last (ninits [x])
  = map last [[x]]
  = [x]
```

Recursive case:

```
I.H.   (map last . ninits) (init xs)
     = init xs

(map last . ninits) xs
    -- Lemma 2
  = ((map last . ninints) xs) ++  [last xs]
    -- I.H.
  = init xs ++ [last xs]
    -- clear from defn of init
  = xs
```

Lemma 2: `map f . ninits = ((map f . ninints) xs) ++ [f xs]`

```
    (map f . ninits) xs
    -- defn. of .
  = map f (ninits xs)
    -- defn. of ninits
  = map f (ninits (init xs) ++ [xs])
    -- map f (xs ++ ys) = map f xs ++ map f ys
  = map f (ninits (init xs)) ++ [f xs]
    -- introduce .
  = ((map f . ninints) xs) ++  [f xs]
```

□

2: `counit . extend f = f`

Proof.

```
    counit . extend (GM f)
  = (GM last) . (GM map f . ninits)
    -- defn. of . on category of graphs
  = GM (last . map f . ninits)
    -- Lemma 3
  = GM f
```

Lemma 3: `last . map f . ninits = f`

```
    (last . map f . ninits) xs
    -- Lemma 2
  = last (((map f . nininits) xs) ++  [f xs])
    -- defn. of last
  = f xs
```

□

3: `extend f . extend g = extend (f . extend g)`

Proof.

```
    extend (GM f) . extend (GM g)
    -- defn. of extend
  = (GM map f . ninits) . (GM map g . ninits)
    -- defn. of . for category of graphs
  = GM (map f . ninits . map g . ninits)
    -- Lemma 4
  = GM (map (f . map g . ninits) . ninits)
    -- defn. of extend
  = extend (GM (f . map g . ninits))
    -- defn. of . for category of graphs
  = extend ((GM f) . GM (map g . ninits))
    -- defn. of extend
  = extend ((GM f) . extend (GM g))
```

Lemma 4: `map f . ninits . map g) = map (f . map g . ninits)`
Base case:

```
    ninits (map g (ninits [x]))
  = ninits (map g [[x]])
  = ninits [g [x]]
  = [[g [x]]]
  = [(map g) [[x]]]
  = [(map g . ninits) [x]]
  = map (map g . ninits) [[x]]
  = map (map g . ninits) (ninits [x])
```

Recursive case

```
I.H.   ninits (map g (ninits (init xs)))
     = map (map g . ninits) (ninits (init xs))

    (ninits . map g . ninits) xs
    -- defn. of .
  = ninits (map g (ninits xs))
    -- defn. of ninits
  = ninits (init (map g (ninits xs))) ++ [map g (ninits xs)]
    -- init commutes with map
```

```
  = ninits (map g (init (ninits xs))) ++ [map g (ninits xs)]
    -- defn. of ninits
  = ninits (map g (init ((ninits (init xs) ++ [xs]))) ++ [map g (ninits xs)]
    -- defn. of init
  = ninits (map g (ninits (init xs))) ++ [map g (ninits xs)]
    -- I.H.
  = map (map g . ninits) (ninits (init xs)) ++ [map g (ninits xs)]
    -- defn. of map
  = map (map g . ninits) (ninits (init xs) ++ [xs])
    -- defn. of ninits
  = map (map g . ninits) (ninits xs)
    -- defn. of .
  = (map (map g . ninits) . ninits) xs
```

□

6.2.3 The Pebbling comonad

As described in section 3.2 the Pebbling comonad is similar in style to the Ehrenfeucht-Fraïssé comonad but the universe is made of lists of pairs where the first element in the pair specifies the index of the pebble being placed.

Once again, this `CComonad` instance specifies an indexed family of comonads differentiated by an invariant. The invariant is:

$\exists k \in \mathbb{Z}^+$. $\forall$ `Graph A`, `xs` $\in$ (`universe` (`Pebble A`)). `max (map fst xs)` $\leq k$ `& xs != []`

The code of this looks somewhat similar to the Ehrenfeucht-Fraïssé comonad.

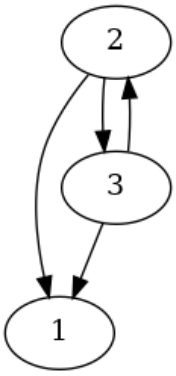
```
data Pebble a where Peb :: (Graph a) =>
    AdjacencyMap [(Int, Vertex a)] -> Pebble a

instance CComonad Pebble GraphMorphism where
    counit        = GM $ snd . last
    extend (GM f) = GM $ map (\xs -> (fst (last xs),f xs)) . ninits
```

We have chosen not to give a full formal proof of correctness as we did for Ehrenfeucht-Fraïssé . The proof follows a similar pattern but is made much longer by the pairs involved. The proof that the pebbling comonad is a valid comonad (as opposed to this implementation of it) is given in [1].

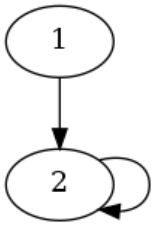
6.2.4 Other Category level concepts

Co/products

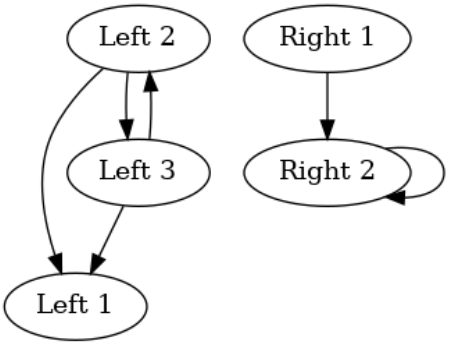
We can included Product and Coproduct bifunctors over the category. These work similarly to the comonads, in that they use a GADTs data constructor to take the `Graph` types to product/coproduct types. The product and coproduct data types are also instances of `Graph`. For both these functors there exists a natural isomorphism from the ID functor. This takes the form of a function that can take two graphs and return the co/product graph. For details see the implementation.



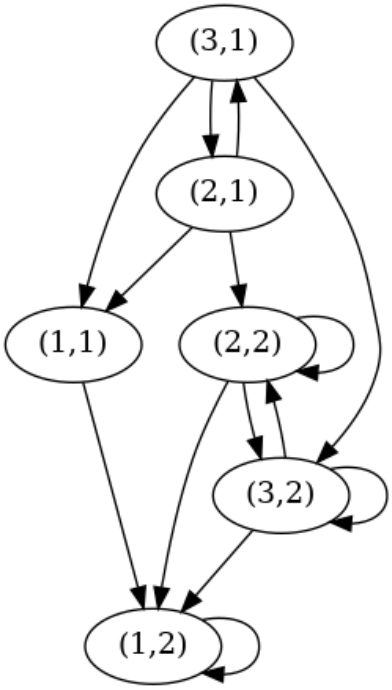
(a) Graph $\mathscr{A}$



(b) Graph $\mathscr{B}$



(c) The Coproduct Graph $\mathscr{A} + \mathscr{B}$



(d) The Product Graph $\mathscr{A} \times \mathscr{B}$

Figure 6.1: Two graphs with their product and coproduct

Co/equalisers

In the implementation there is a function `getEqualizer` with the type given below.

```
getEqualizer :: (Graph a, Graph b) => a -> b ->
        GraphMorphism a b -> GraphMorphism a b -> a
```

As may be clear from the type signature, this is not a true equalizer in the categorical sense. An equalizer, as described in section 2.1 takes two morphisms `f :: a -> b` and `g :: a -> b` and returns a third object `c` and morphism `h :: c -> a` such that `h . f = h . g` and `c` is universal.

The function in question takes two `GraphMorphism` between the same two graphs and builds the sub-graph of the first graph where all the vertices are mapped to the same vertex in the second graph by the two `GraphMorphism` functions.

In the category $\mathcal{R}(\sigma_G)$ the objects are models where the `Vertex` type determines the universe. For the equalizer we should construct a new return type `c` consisting of only the elements of $x \in$ `a` such that `f` $x =$ `h` x. Sadly there is no easy way in Haskell to build new types and use them as a return value. This would require a dependently typed language such as Idris.

Instead, we have used `a` as the return type since `c` should be a subtype of `a`. The graph returned is the correct shape but doesn't have the required `Vertex` type.

Coequalizers are yet more problematic. We need to create a new quotient type given by the equivalence relation `f` $x \sim$ `g` x. Again this is not possible in Haskell. For this reason no `getCoequilzer` function is given.

6.2.5 Building Ehrenfeucht-Fraïssé and Pebbling graphs

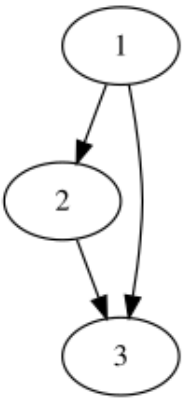
We now have a category of graphs with two comonad and several other functors implemented over it. The comonads are useful on their own but to actually check that the Duplicator has a valid strategy we need to check the validity of the homomorphism $\mathbb{E}_k\mathscr{A} \to \mathscr{B}$. To check this we first need to generate the graph $\mathbb{E}_k\mathscr{A}$. Luckily there is a natural transformation from the ID functor to $\mathbb{E}_k$. This means we can define a function `graphToEFk` that takes a graph $\mathscr{A}$ and returns the graph representing all possible Spoiler plays of length k, $\mathbb{E}_k\mathscr{A}$.

The function generates all possible plays then inserts the valid plays into the graph in the appropriate edges. This is necessarily not an efficient operation. In fact the complexity of the function is around $\mathcal{O}(k \cdot \frac{|VE|!}{(|VE|-k)!})$ where VE is the set of vertices in at least one edge of the graph. For the sake of brevity we will omit the full code here.

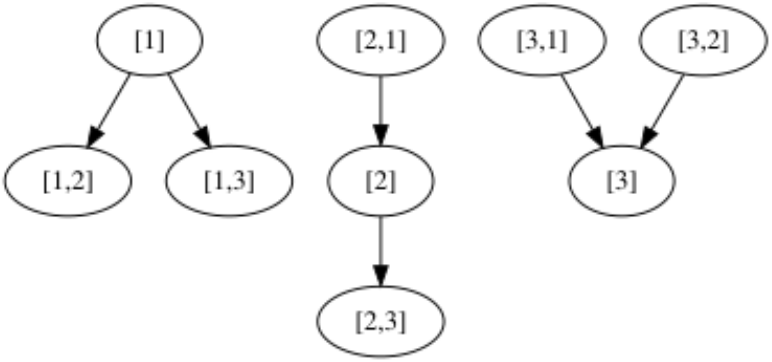
For pictorial representation of the graphs generated see figures 6.2 and 6.3. It is clear from the images just how quickly the size of the graph grows.

There is also a function `graphToLimEFk` that constructs a bounded size a sub-graph of the $\mathbb{E}_k$ graph for testing. This is a lot quicker to generate if the bound is reasonable.

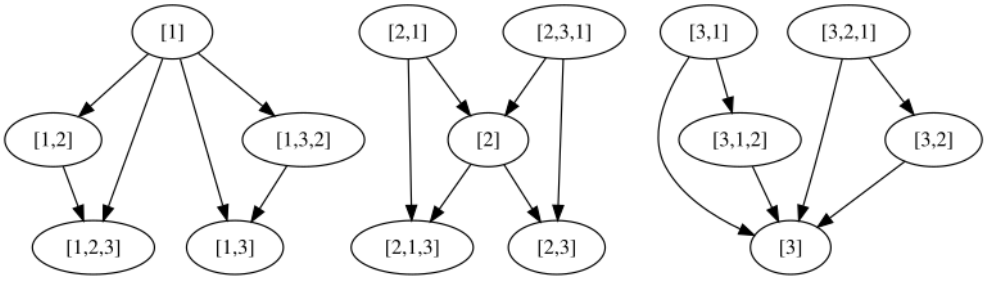
With these tools we can now build the $\mathbb{E}_k$ graph of arbitrary graphs. The next logical step would be to write a function to build $\mathbb{P}_k$ graphs. Unfortunately there are some issues with this. Since the resource limit on pebbling games is the number of pebbles and pebbles can be moved, the length of plays in a $\mathbb{P}_k$ graph has no limit. Moreover, the graph is infinite. It would be possible to generate a finite subgraph, however it would only allow for proofs up to the same level of logical equivalence as the $\mathbb{E}_k$ morphisms [6].



(a) Linear order graph $\mathscr{A}$



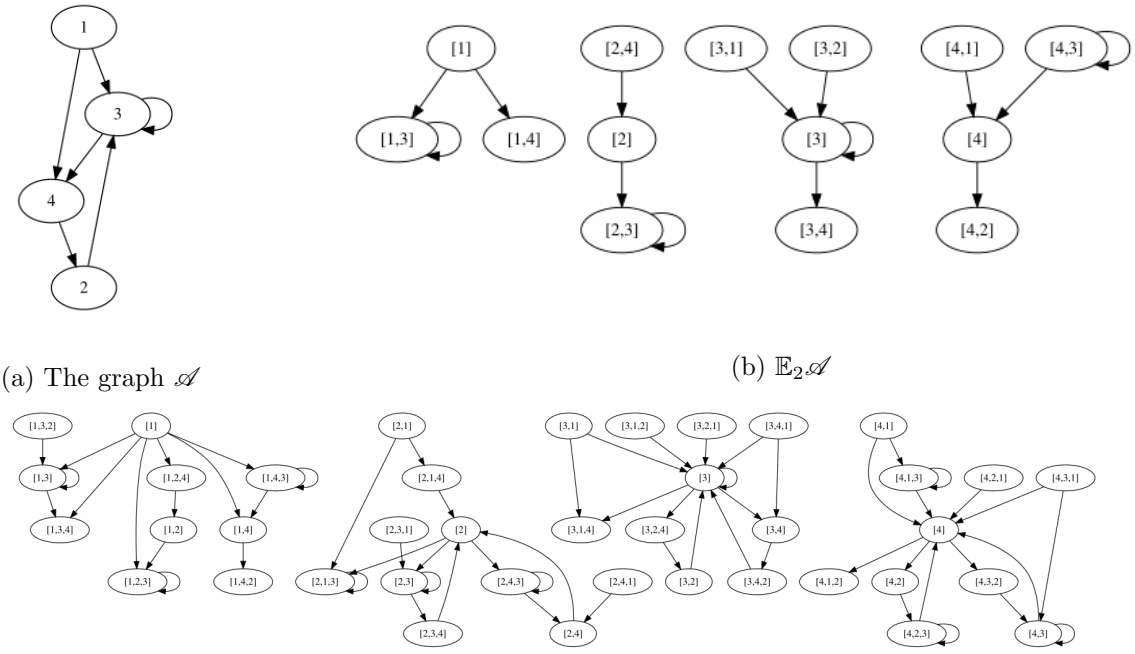
(b) $\mathbb{E}_2\mathscr{A}$



(c) $\mathbb{E}_3\mathscr{A}$

(d) $\mathbb{E}_4\mathscr{A}$

Figure 6.2: The graph representation of the linear order $[1, 2, 3]$ and its mapping under the $\mathbb{E}_2$, $\mathbb{E}_3$ and $\mathbb{E}_4$ comonads.



(a) The graph $\mathscr{A}$

(b) $\mathbb{E}_2\mathscr{A}$

(c) $\mathbb{E}_3\mathscr{A}$

Figure 6.3: A randomly generated graph and its mapping under the $\mathbb{E}_2$ and $\mathbb{E}_3$ comonads

6.2.6 Checking validity

Building the graph $\mathbb{E}_k\mathscr{A}$ for some graph $\mathscr{A}$ is not the only use of $\mathbb{E}_k$ graphs. We may also want to check that a homomorphism takes a graph $\mathscr{A}$ to a valid $\mathbb{E}_k\mathscr{A}$ graph, for instance when showing that a coalgebra is valid. For this reason we have functions `checkMorphIsHomo`, `checkValidEFkGraph` and `checkValidPebkGraph`.

The function `checkMorphIsHomo` does what it says on the tin. Given two graphs `a` and `b` and a `GraphMorphism` function `gm`, it checks that `gm` is a valid homomorphism from `a` to `b`. Note that the homomorphism `gm` is valid from `a` to the graph `apply gm a`, however it is only valid form `a` to `b` if `apply gm a` is a sub-graph of `b`.

The functions `checkValidEFkGraph` and `checkValidPebkGraph` both fold over the edges of the graphs checking that all the conditions given in chapter 3 for the graphs $\mathbb{E}_k\mathscr{A}$ and $\mathbb{P}_k\mathscr{A}$ hold. For the full implementation see the code.

6.2.7 Proof checkers

Now we come on to the most important functions in the implementation: the functions that verify that models are equal in specified logics using the isomorphisms between certain morphisms and logical equivalence. As discussed in section 3.2.1 if there exist coKleisli morphisms $\mathbb{E}_k\mathscr{A} \to \mathscr{B}$ and $\mathbb{E}_k\mathscr{B} \to \mathscr{A}$ then $\mathscr{A} \equiv^{\exists\mathcal{L}^k} \mathscr{B}$, and if they form an isomorphism then $\mathscr{A} \equiv^{\mathcal{L}^k(\#)} \mathscr{B}$. We have a function to show each of these. `eqQRankKEPfrag` returns `true` if there is equality in $\exists\mathcal{L}^k$ between the two graphs and `eqQRankKCounting` returns `true` if there is equality in $\mathcal{L}^k(\#)$.

Haskell is not a language designed for proof checking. Since it is Turing complete, it is not possible to fully verify code. We have to rely on invariants and assertions about the nature of the inputs to a function to draw conclusions about the output. Therefore, we do not claim that these functions constitute formal proof, however, they are about as

close as it is possible to get in Haskell.

The definitions of the functions are as follows:

```
eqQRankKEPfrag :: (GraphCoerce a, GraphCoerce b) =>
    Int -> CoKleisli EF GraphMorphism a b ->
    CoKleisli EF GraphMorphism b a -> a -> b -> Bool
eqQRankKEPfrag k (CoKleisli h1) (CoKleisli h2) g1 g2 =
    (isSubgraphOf (gcoerce (apply h1 (graphToEFk k g1))) (gcoerce g2)) &&
    (isSubgraphOf (gcoerce (apply h2 (graphToEFk k g2))) (gcoerce g1))

eqQRankKCounting :: (GraphCoerce a, GraphCoerce b) =>
    Int -> CoKleisli EF GraphMorphism a b ->
    CoKleisli EF GraphMorphism b a -> a -> b -> Bool
eqQRankKCounting k (CoKleisli h1) (CoKleisli h2) g1 g2 =
    (apply h1 (graphToEFk k g1) == g2) &&
    (apply h2 (graphToEFk k g2) == g1)
```

As can be seen here, these functions make use of the `graphToEFk` function meaning their complexity is high. Since there is also no equivalent function for the pebbling comonad there is no way to implement proof checkers for it.

As discussed in section 3.2.1 there are characterizations of the plain $\mathcal{L}^k$ and $\mathcal{L}_k$ logics using spans that are given in [2]. Due to the limited scope and time constraints on this project these were not explored in Haskell. They would, however, make a valuable extension to the work of this paper.

6.3 Coalgebras

The final theoretical contribution this project makes in Haskell are to do with the coalgebras of $\mathbb{E}_k$. As discussed in section 3.1.1 the coalgebras of the $\mathbb{E}_k$ comonad are equivalent to tree depth decompositions of the Gaifman graph of depth k. We have implemented functions that make use of this isomorphism.

The functions `forestToCoalg` and `coalgToForest` convert between the tree depth decomposition (as a forest) and coalgebras, these follow the ideas of the proof in [2]. There is also function to check if a morphism $\mathscr{A} \to \mathbb{E}_k\mathscr{A}$ is a valid $\mathbb{E}_k$ coalgebra on $\mathscr{A}$ `checkValidCoalg`. This works by converting the coalgebra to a tree depth decomposition and checking the validity of the tree depth decomposition over the graph with another function `checkValidTDD`.

In the function `coalgToForest` the first step when converting the coalgebra to its forest decomposition is to generate the Gaifman graph of the coalgebras domain. In section 2.2.3, we mentioned that the Gaifman graph of any graph is its irreflexive undirected closure, so this is exactly how we build it. The most natural way to implement this function is an object map, similar to that of `EF`. This also allows us to write functions that restrict the type to only Gaifman graphs however we do not have any type level guarantees that it truly is a Gaifman graph.

```
data Gaifman a where Gaifman :: (Graph a) => AdjacencyMap (Vertex a) -> Gaifman a

getGaifmanGraph :: AdjacencyMap a -> Gaifman (AdjacencyMap a)
getGaifmanGraph g = Gaifman $ edges (nub (concat newedges))
    where newedges = [[(v1,v2),(v2,v1)] |(v1,v2) <- (edgeList g), v1 /= v2]
```

We have also implemented a function to get the tree depth decomposition of a graph for testing purposes, `gaifmanTDD`. See the implementation for details.

To generate the coalgebra $\mathscr{A} \to \mathbb{E}_k\mathscr{A}$ from the tree depth decomposition of $\mathscr{A}$ we construct a function that, given an element $a \in A$, finds the tree in the forest containing a and returns the list of its predecessors up to the root (using the function `findNodePreds` given in the implementation). This is the action of the function `forestToCoalg`.

To go the other way is a bit more involved. To start with, to get the codomain graph of the coalgebra we need to know the exact shape of the domain graph, so we need to take it as an argument to our function:

```
coalgToForest :: (GraphCoerce a) => GraphMorphism a (EF a) ->
                      a -> Forest (Vertex a)
```

We then need to apply the coalgebra to every element in the graph to get a list. Some of these lists will be prefixes of others already generated, others will be extensions of an already existing list. We want to end up with only the maximal lists in this prefix ordering. The methods of generating these maximal lists are too long to describe here. We refer the reader to the implementation for details.

As well as these functions for converting to and from coalgebras we also have functions that check the validity of a coalgebra or tree depth decomposition. The function `checkValidTDD :: (GraphCoerce a) => a -> Forest (Vertex a) -> Bool` checks the validity of the decomposition (every edge connected in the graph should be in the same tree in the decomposition). The function `checkValidCoalg` converts the coalgebra to a tree depth decomposition then uses the function `checkValidTDD`. This makes use of the isomorphism between the coalgebras and the decompositions.

Chapter 7

Examples

The primary use of Ehrenfeucht-Fraïssé and Pebbling games is to prove inexpressibility of a property in a given logic. This can be done by giving two σ-structures one which satisfies the property and another which does not, then proving them equal using some variant of the game.

To prove two models $\mathscr{A}$ and $\mathscr{B}$ equal in the quantifier rank k fragment of first order logic, $\mathcal{L}^k$, we construct two homomorphisms $\mathbb{E}_k\mathscr{A} \to \mathscr{B}$ and $\mathbb{E}_k\mathscr{B} \to \mathscr{A}$. If these form an isomorphism then they are equal in $\mathcal{L}^k(\#)$, otherwise, in $\exists\mathcal{L}^k$. In [2] it is show that the existence of these morphisms is equivalent to a winning strategy for Duplicator. The proof is constructive, so given a Duplicator strategy (with type signature `(Graph a, Graph b) => Int -> a -> b -> [Vertex a] -> [Vertex b]`) we can build a coKleisli morphism. We have thus implemented a function `buildMorphFromStrat` which takes a Duplicator strategy and returns a homomorphism between two graphs.

```
buildMorphFromStrat :: (GraphCoerce a, GraphCoerce b) =>
    (Int -> a -> b -> [Vertex a] -> [Vertex b]) -> Int -> a -> b ->
    CoKleisli EF GraphMorphism a b
buildMorphFromStrat strat k glin1 glin2 =
    CoKleisli $ GM (last . strat k glin1 glin2)
```

7.1 Proof of concept example

We start off with a simple example that takes two isomorphic graphs and an isomorphism between them and shows that they are equal. It generates a random graph, `graph1` then uses the isomorphism to generate the second graph, `graph2`. The function `liftMapToEFMorph` takes a mapping from one graph to the other and builds a coKleisli morphism.

```
proof1 :: Int -> Int -> Int -> (Int -> Int, Int -> Int) -> Bool
proof1 seed size k iso = eqQRankKCounting k g1tog2 g2tog1 graph1 graph2
    where graph1 = getRandomGraph seed size
          graph2 = gmap (fst iso) graph1
          g1tog2 = liftMapToEFMorph (fst iso)
          g2tog1 = liftMapToEFMorph (snd iso)

liftMapToEFMorph ::(Graph a, Graph b) => (Vertex a -> Vertex b)
    -> CoKleisli EF GraphMorphism a b
liftMapToEFMorph f = CoKleisli $ GM f Category.. counit
```

We can, for example, run something like the function `ex1 = proof1 45 3 4 ((\x -> x*2), (\x -> x 'div' 2))`. If this returns true (which it always should), this constitutes a proof that the two isomorphic graphs are indistinguishable in $\mathcal{L}^k(\#)$. The function `proof2` (see the implementation) shows that they are equal in $\exists\mathcal{L}^k$, though this is implied by `proof1`.

7.2 Linear Order Example

Our first comprehensive example is based on the proof of Theorem 3.6 in the book Finite Model Theory by Libkin [16].

Theorem. *Let $k > 0$, and let L_1 and L_2 be linear orders of length at least 2^k. Then $L_1 \equiv_{\mathcal{L}^k} L_2$.*

In the context of the Ehrenfeucht-Fraïssé comonad this theorem says that, if we have two linear orders L_1 and L_2 satisfying the conditions above, it should be possible to construct two morphisms $\mathbb{E}_k\mathscr{A}_{L_1} \to \mathscr{A}_{L_2}$ and $\mathbb{E}_k\mathscr{A}_{L_2} \to \mathscr{A}_{L_1}$ where $\mathscr{A}_L$ is the linear order L represented as a graph with the edge relation as $<$.

For the full proof see [16]. Here we present a constructive version of it that allows us to take a Spoiler strategy and generate the Duplicators response. From this we can build the coKleisli morphisms using the function `buildMorphFromStrat`.

7.2.1 Haskell Linear Order Proof

By $L^{\leq a}$ we mean the substructure of L that consists of all the elements $b \leq a$ and by $L^{\geq a}$ we mean the substructure of L that consists of all the elements $b \geq a$. In this proof the Spoiler plays in the linear order L_1 and the Duplicator in the linear order L_2. To begin, we prove the following lemma:

Lemma. *Let $L_1, L_2, a \in L_1$ and $b \in L_2$ be such that*

$$L_1^{\leq a} \equiv_{\mathcal{L}^k} L_2^{\leq b} \text{ and } L_1^{\geq a} \equiv_{\mathcal{L}^k} L_2^{\geq b}$$

Then $L_1 \equiv_{\mathcal{L}^k} L_2$ where $a \equiv b$.

The proof for the lemma is simple, if the Spoiler plays in $L_1^{\leq a}$ use the winning strategy for $L_2^{\leq b}$ (which must exists because $L_1^{\leq a} \equiv_{\mathcal{L}^k} L_2^{\leq b}$) and if they play in $L_1^{\geq a}$ use the winning strategy for $L_2^{\geq b}$.

For the proof of the theorem itself we use induction on k. Suppose the Spoiler plays $a \in L_1$, then the proof gives $b \in L_2$ such that $L_1 \equiv_{k-1} L_2$ with $a \equiv b$. Haskell excels at proof by induction, once we find b such that $a \equiv b$ we can fix it, then call the proof again on the linear orders with $a \equiv b$ and $k - 1$ and with the Spoilers play one shorter.

The code for generating the Spoilers play primarily consists of two functions `linRec` and `lemma`, there are numerous auxiliary functions, but we will not present them in detail. We give a description of the functionality of `linRec` and `lemma`, however, for the sake of brevity rather than present the code we will refer the reader to the implementation.

The function `linRec :: Int -> [a] -> [b] -> [a] -> [(a,b)] -> [(a,b)]` takes five arguments, `k`, `lin1`, `lin2`, `spoilerplay` and `iso` and returns a partial isomorphism between `lin1` and `lin2` including all elements in `spoilerplay`. The linear orders `lin1` and `lin2` are converted back into lists from graphs to make this function more manageable. `iso` is a list of pairs containing all the current know mappings, it starts off empty but as

we fix elements $a \equiv b$ we add them to `iso`. The function recurses over `spoilerplay` and `k`.

The function itself is split into four cases. The variable `a` is the Spoilers play, the function finds an element `b` that it can add to `iso` and recurses.

1. The head of `spoilerplay` is already in `iso` - in this case, simply recurse with $k-1$.
2. $\texttt{lin1}^{\leq a}$ has length less than 2^k - here we pick `b` as the element `lin2 !! (index a lin1)`, then add `zip` $\texttt{lin1}^{\leq a}$ $\texttt{lin2}^{\leq b}$ to `iso` and recurse with $\texttt{lin1}^{\geq a}$ and $\texttt{lin2}^{\geq b}$ as the new linear orders.
3. $\texttt{lin1}^{\geq a}$ has length less than 2^k - here we pick `b` as the element $\texttt{lengthlin1}^{\geq a}$ from the end of `lin2`, then add `zip` $\texttt{lin1}^{\geq a}$ $\texttt{lin2}^{\geq b}$ to `iso` and and recurse with $\texttt{lin1}^{\leq a}$ and $\texttt{lin2}^{\leq b}$ as the new linear orders.
4. $\texttt{lin1}^{\geq a}$ and $\texttt{lin1}^{\leq a}$ both have length greater than 2^k - this is where we use `lemma`. The `lemma` function splits the Spoiler play into the parts that are played on $\texttt{lin1}^{\geq a}$ and $\texttt{lin1}^{\leq a}$ then calls `linRec` on both of these. We select `b` here as `lin2 !! (2**(k-1))`

We can then run `proof3` with two linear orders to prove they are equal in the existential positive fragment with quantifier rank k:

```
proof3 k lin1 lin2 = eqQRankKEPfrag k gm1 gm2 g1 g2
    where gm1 = buildMorphFromStrat linStrategy k g1 g2
          gm2 = buildMorphFromStrat linStrategy k g2 g1
          g1 = linToGraph lin1
          g2 = linToGraph lin2
```

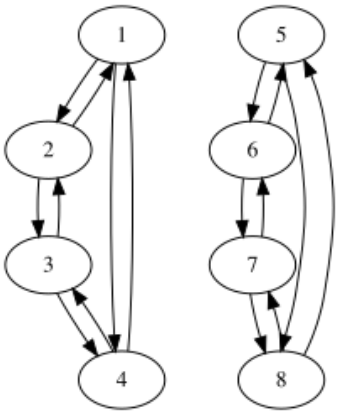
The proof function builds the appropriate morphisms then uses `eqQRankKEPfrag` to check the equality.

7.3 Undefinability of Two Colourable/Even/Connected

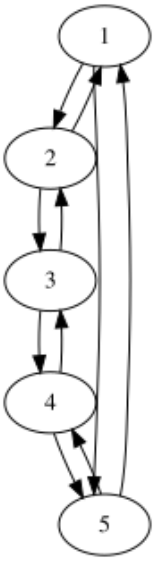
For the second example we prove that certain property in undefinable in the existential fragment first order logic with a finite quantifier rank. We find two graphs such that they can be proved equivalent in the k variable game where one satisfied some property and the other does not.

It so happens that the graph shown in figure 7.1a, $\mathscr{A}$, is two colourable, even and disconnected whilst the graph in figure 7.1b, $\mathscr{B}$, is connected but not even or two colourable. We can generalize these two graphs, the first to a single cycle of size $2^k + 1$, $\mathscr{A}_k$, and the second, $\mathscr{B}_k$, made up of two cycles each of size 2^k. Hence, if we can prove the existence of homomorphisms $\mathbb{E}_k\mathscr{A}_k \to \mathscr{B}_k$ and $\mathbb{E}_k\mathscr{B}_k \to \mathscr{A}_k$ then we have a proof that these properties are not definable in $\exists\mathcal{L}^k$. This is exactly what we have done. The proof method we used is based on methods in [7] and slides by Anuj Dawar [6]. We first need to generate the graphs. This is done by the function `generateCycleG :: Int -> (AdjacencyMap Int, AdjacencyMap Int)` which takes a parameter k and returns the two graphs.

Given a Spoiler play on one of the graphs we need to generate a Duplicator play. The strategy is actually independent of the graph the Spoiler chooses to play on. This is because it is designed for Ehrenfeucht-Fraïssé games where the Spoiler can play on either graph, however, we only have the capability to prove results where the Spoiler plays on one of the graphs.



(a) Two cycles of length 2^k



(b) A cycle of length $2^k + 1$

Figure 7.1: The two graphs where $k = 2$

The Duplicator's strategy is to ensure that the distance between any two of her pebbles is either equal to the distance between the corresponding pair of the Spoilers pebbles, or, it is greater than $2^k - r$ where r is the round number, starting from 1. This strategy ensures that the Spoiler can never force Duplicator into a position where she is forced into revealing that two nodes that should be adjacent are not or vice versa.

The actual code for this strategy amounts to a lot of searching round the graphs to find a placement for the nodes that preserves the properties. I will not present the full code here because it will not greatly add to the readers understanding. Please refer to the implementation for the code.

The function `cycStrategy` implements the actual strategy. The function `proof5` takes an argument `k` and proves that the two graphs are equal in $\exists\mathcal{L}^k$.

```
proof5 k = eqQRankKEPfrag k gm1 gm2 g1 g2
    where gm1 = buildMorphFromStrat cycStrategy k g1 g2
          gm2 = buildMorphFromStrat cycStrategy k g2 g1
          (g1,g2) = generateCycleG k
```

We cannot prove these two graphs equal in $\mathcal{L}^k(\#)$ (with the `eqQRankKCounting` proof checker). In fact the formula $\exists_{2^k+2}x$ is true for the first graph but not the second.

7.4 Undefinability of Hamiltonian

A Hamiltonian cycle is a cycle in a graph that visits each vertex of the graph exactly once. It is provable that it is impossible to axiomatize the property of Hamiltonicity in a finite number of variables in FOL. We can do this with pebble games by proving that two graphs $\mathscr{A}_k$ and $\mathscr{B}_k$, one of which has a Hamiltonian cycle and the other not, are equivalent in the k pebble game. Therefore, for no finite k is there a k variable formula that captures Hamiltonicity. Hence, the property is not axiomatizable in FOL.

With our implementation we cannot prove this. We would need a function `graphToPebK` that generated pebbling graphs, unfortunately, since this is infinite (as discussed in section

6.2.5) we cannot implement this easily in Haskell.

Instead, we have provided the morphisms `hamMorphs`, which theoretically can prove the result, though there is no way to check this computationally.

The two graphs we use are bicliques. The first of graphs has k nodes on both sides. The second has k nodes on the left and $k+1$ node on the right. The first of these admits a Hamiltonian cycle, the second does not.

The Spoiler plays by placing his pebbles on one of the graphs. The Duplicator's strategy is to simply place her pebble on the same side of the other graph. It doesn't matter which node she chooses since all the nodes on a given side are indistinguishable. The graph has k nodes on each side so the Duplicator will always be able to find a node on the same side as the Spoiler.

This strategy is fairly straight forward, we refer the reader to the implementation for more details.

The code for the morphisms themselves is given below:

```
hamMorphs :: (Graph b, Graph a, Vertex a ~ Int, Vertex b ~ Int) =>
    Int -> (CoKleisli Pebble GraphMorphism a b,
    CoKleisli Pebble GraphMorphism b a)
hamMorphs k = (CoKleisli $ counit . GM (hamStrategy k),
               CoKleisli $ counit . GM (hamStrategy k))
```

If a function to generate Pebble graphs were to be implemented, these morphisms could be checked. This would constitute a proof that Hamiltonicity is not definable in FOL.

Chapter 8

Conclusions

In this paper we have given Haskell implementations of two game comonads from [2] along with function to check equivalence proofs in the associated logics. For a full overview of the implementation see chapter 5.

8.1 A Reflection on Design Decisions

In the initial development of this project, the first design decision was the choice of functional language for the implementation. Haskell was chosen because of its wide adoption and strong integration of categorical concepts as well as the author's familiarity with the language. It does, however, have its drawbacks. Haskell does not have a mature dependent types framework and the language does not naturally support proof checking as it is Turing complete.

During the development, we found the drawbacks of Haskell turned out to be rather limiting. The functions that "prove" certain properties are inexpressible cannot be validated in the formal sense. Instead, they are checks up to a given degree of confidence. On top of this, working in a dependently typed language would have greatly simplified development. For example, when implementing the category $\mathcal{R}(\sigma)$, the signature was restricted to σ_G. In a language with dependent type a category could be naturally parameterized by a signature. We could also have restricted the universe of the comonads at the type level, rather than relying on invariants as we have.

If further research was to be carried out on programmatically implementing game comonads then the author of this paper recommends using a dependently typed language designed for proof checking such as Idris or Agda (see section 4.1.1). Both of these languages are sufficiently similar to Haskell that code from this project could be translated and adapted.

8.2 Extensions

There are numerous ways in which this project could be extended to encompass more of the computational power of the game comonads.

8.2.1 Modal comonad

Along with the Ehrenfeucht-Fraïssé and Pebbling comonad a third comonad was presented in [2], the modal comonad. This provides categorical semantics to bisimulation games between Kripke structures. Like the Ehrenfeucht-Fraïssé game the length of the plays are

limited by a resource parameter k so the implementation should not suffer from the same problems as the Pebbling comonad.

8.2.2 Spans

In Ehrenfeucht-Fraïssé and Pebbling games over two structures $\mathscr{A}$ and $\mathscr{B}$ the Spoiler can play in both structures. A homomorphism $\mathbb{C}_k\mathscr{A} \to \mathscr{B}$ only captures games where the Spoiler plays only in $\mathscr{A}$ and the Duplicator only in $\mathscr{B}$ and $\mathbb{C}_k\mathscr{B} \to \mathscr{A}$ and thus can only prove the structures equivalent in $\exists\mathcal{L}$ or $\mathcal{L}(\#)$. Ideally we would like to capture the full game.

The comonads suffer from the goldilocks problem: the logics are either too expressive or not enough, never just right. In [2] a solution to this problem is given provided using the categorical concept of spans. An implementation of these could be explored to allow for proofs of equivalence in less restricted logics.

8.2.3 Coalgebra categories

The rank k coalgebras of both the Ehrenfeucht-Fraïssé and Pebbling comonads are isomorphic to tree depth/width decompositions of depth/width k respectively. The category of these tree decompositions is isomorphic to the category of coalgebras (the Eilenberg-Moore category) for each of the comonads. The coalgebras of the both of comonads are closed under colimits. We could therefore give constructions such as coequalizers and coproducts on this category, and use them to compose the coalgebras/tree decompositions. This would allow for novel ways of generating tree decomposition of graphs via compositionality of coalgebras.

8.2.4 Adjunction

All co/monads arise from adjunction. The Ehrenfeucht-Fraïssé comonad arises from adjunction between the category of tree depth decompositions of height k and $\mathcal{R}(\sigma)$. The pebbling comonad arises from the adjunction between the category of tree width decompositions of size k and $\mathcal{R}(\sigma)$, hence the isomorphism between coalgebras and these decompositions. Work was started on expressing the comonads via these adjunctions in Haskell however due to time constrains never finished.

8.2.5 Algebraic Graphs

The Haskell `algebraic-graphs` library (section 4.2.1) provides a method of constructing arbitrary graphs using two operations, `overlay` and `connect`. Every graph is associated with a tree decomposition. The coalgebras, and hence the tree width/depth decompositions are closed under colimits. This suggests that it is possible build the tree decomposition alongside the graph, merging the tree width/depth decompositions of two graphs joined by the `connect` and `overlay` functions by finding the equivalent combinators on the category of tree decompositions.

This would allow for a static way to build provably correct tree decompositions at compile time rather than runtime. Importantly, in use cases where graphs are built via composition this provides an efficient way to build the tree decomposition whilst keeping track of the bound k.

Bibliography

[1] Samson Abramsky, Anuj Dawar, and Pengming Wang. The pebbling comonad in finite model theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[2] Samson Abramsky and Nihil Shah. Relating structure and power: Extended version. *CoRR*, abs/2010.06496, 2020.

[3] Andrej Bauer. Hask is not a category. http://math.andrej.com/2016/08/06/hask-is-not-a-category/, 2016.

[4] Hans L Bodlaender. Discovering treewidth. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 1–16. Springer, 2005.

[5] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977.

[6] Anuj Dawar. Descriptive complexity. www.cl.cam.ac.uk/ ad260/talks/caleidoscope1.pdf, 2019.

[7] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Springer Science & Business Media, 2005.

[8] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*, 49:129–141, 1961.

[9] Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM J. Comput.*, 28(1):57–104, 1998.

[10] Roland Fraïssé. *Sur quelques classifications des systemes de relations*. PhD thesis, Imprimerie Durand, 1955.

[11] Shauna CA Gammon. *Notions of category theory in functional programming*. PhD thesis, University of British Columbia, 2007.

[12] Fabrizio Genovese, Alex Gryzlov, Jelle Herold, Andre Knispel, Marco Perone, Erik Post, and André Videla. idris-ct: A library to do category theory in idris. In John Baez and Bob Coecke, editors, *Proceedings Applied Category Theory 2019, ACT 2019, University of Oxford, UK, 15-19 July 2019*, volume 323 of *EPTCS*, pages 246–254, 2019.

[13] Jason Z. S. Hu and Jacques Carette. Formalizing category theory in agda. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 327–342. ACM, 2021.

[14] Yoichi Iwata, Tomoaki Ogasawara, and Naoto Ohsaka. On the power of tree-depth for fully polynomial FPT algorithms. *CoRR*, abs/1710.04376, 2017.

[15] P.G. Kolaitis and M.Y. Vardi. On the expressive power of datalog: Tools and a case study. *Journal of Computer and System Sciences*, 51(1):110–134, 1995.

[16] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[17] Andrey Mokhov. Algebraic graphs with class (functional pearl). In Iavor S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 2–13. ACM, 2017.

[18] Dominic Orchard and Alan Mycroft. Categorical programming for data types with restricted parametricity. *Draft submitted and rejected by TFP*, 2012.

[19] Dominic Orchard and Alan Mycroft. A notation for comonads. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.

Appendices

Appendix A

The Implementation

A.1 The Category Framework

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE AllowAmbiguousTypes #-}

module Category where

import Data.Kind
import Data.Coerce
import Data.Functor.Identity
import Prelude hiding ((.))

class Category (cat :: k -> k -> *) where
  type Object cat (o :: k) :: Constraint
  id :: (Object cat o) => cat o o
  (.) :: (Object cat a, Object cat b, Object cat c) => cat b c -> cat a b -> cat a c

class (Category dom, Category cod) => CFunctor f dom cod where
  funcMap :: (Object dom a, Object cod (f a), Object dom b, Object cod (f b)) => dom a b -> cod (f a) (f b)

class (Category dom1, Category dom2, Category cod)
  => CBifunctor f dom1 dom2 cod where
  bifuncMap :: (Object dom1 a, Object dom1 b,
   Object dom2 c, Object dom2 d,
   Object cod (f a c), Object cod (f b d))
   => dom1 a b -> dom2 c d -> cod (f a c) (f b d)

-- Definition of a comonad in coKleislie form
class (Category cat) => CComonad f cat where
  counit :: (Object cat a
    , Object cat (f a))
    => cat (f a) a
  extend :: (Object cat a
    , Object cat b
    , Object cat (f a)
    , Object cat (f b))
    => cat (f a) b
    -> cat (f a) (f b)
  duplicate :: (Object cat a
```

```
    , Object cat (f a)
    , Object cat (f (f a)))
    => cat (f a) (f (f a))
  duplicate = extend Category.id


class (Category dom, Category cod, CFunctor l dom cod, CFunctor r cod dom) => Adjunction cod dom l r where
  eta :: (Object dom a, Object dom (l (r b))) => dom a b

-- Every comonad is automatically a functor
instance (Category cat, CComonad f cat) => CFunctor f cat cat where
  funcMap f = extend (f . counit)

data CoKleisli f cat a b where
  CoKleisli :: (Category cat, CComonad f cat) => cat (f a) b -> CoKleisli f cat a b

instance Category (CoKleisli f cat) where
  type Object (CoKleisli f cat) a = (Category cat, CComonad f cat, Object cat a, Object cat (f a))
  id = CoKleisli counit
  (CoKleisli f) . (CoKleisli g) = CoKleisli (f . extend g)
```

A.2 The Category of Graphs and the Comonads

```
{-# LANGUAGE ConstraintKinds        #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE UndecidableInstances   #-}
{-# LANGUAGE StandaloneDeriving     #-}
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE TypeOperators          #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE AllowAmbiguousTypes    #-}
{-# LANGUAGE InstanceSigs           #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE RankNTypes             #-}
{-# LANGUAGE KindSignatures         #-}


module AlgGraphsCat where

import Algebra.Graph.Class
import Algebra.Graph.AdjacencyMap as AdjMap
import qualified Algebra.Graph.NonEmpty.AdjacencyMap as NonEmptyAdjMap (fromNonEmpty, overlay, edge, edgeList)
import Algebra.Graph.AdjacencyMap.Algorithm

import GHC.TypeLits
import GHC.TypeLits.Extra
import GHC.Exts (Constraint)
import Data.List
import Data.Maybe
import Data.Coerce
import Data.Tree
import Data.Proxy
import Debug.Trace
import Category


-- EF and Pebble comonad over graphs using algebraic adjaceny map graphs.

-- The category of graphs is a category or relational structures over the signitue (edge :: Arity 2)
-- In this program graphs have an underlying vertex type, the elements of this type are taken to be the
-- universe of the relational structure. This means that we consider graphs equal up to the set of edges,
-- i.e. we ignore vertices of degree 0.
```

```haskell
---------- The category of graphs ----------

-- Pre: Any graph that implements Graph must merge vertecies of the same value.

-- Morphisms in the categroy of Graphs

-- Ideally these would be graph homomorphims, however, implementing type level graph homomorphisms in haskell
-- is not easy (or maybe not even possible). It would be easy enough to represent them as a pair of graphs and
-- a mapping between them the could easily be checked however in that case the two graphs would have to be known
-- when constructing the morphism.
-- Instead, I have chosen to represen   t them as vertex maps. A graph morphism can be used to generate the graph of
-- codomain by applying the map to all vertices of the graph and preserving the edge structure. This is
-- nececerraly a subset of graph homomorpisms. Graphs are respresented as adjacency maps from the Algebraic graphs
-- library, these merge vertices with the same value, so the number of edges of the codomain graph is less than
-- or equal to that of the domain graph.
-- This cannot, for example, represent that graph homomorpism between a unconnected and connected graph of a given
-- set of vertices
data GraphMorphism a b where GM :: (Graph a, Graph b) => (Vertex a -> Vertex b) -> GraphMorphism a b

instance Category GraphMorphism where
    type Object GraphMorphism g = Graph g
    id          = GM Prelude.id
    GM f . GM g = GM (f Prelude.. g)

-- Ideally the objects of the category would be any type that implements Graph, however the Algebraic
-- Graphs Graph class only gives functions for construction of graphs, not deconstruction. Because of
-- this it is useful to have a concrete datatype for graphs. Ideally this would not be needed however
-- without depenent types this becomes very difficult for reasons outlined in other comments.
-- AdjacencyMaps have been chosen becuase they are an efficient representation that comes with a
-- useful set of algorithms for graph manipulation.
-- GraphCoerce provides a way of transforming a graph that has been acted on by a functor back into
-- an adjacency map, thus allowing these functions to be used on it.
class Graph g => GraphCoerce g where
    gcoerce    :: g -> AdjacencyMap (Vertex g)
    gcoerceRev :: AdjacencyMap (Vertex g) -> g

-- We need the Ord here because the Graph instance for AdjMaps needs it for a lot of useful commands
-- e.g. vertexList, edgeList...
instance Ord a => GraphCoerce (AdjacencyMap a) where
    gcoerce    = Prelude.id
    gcoerceRev = Prelude.id

-- The EF and Pebble comonads arnt really a functors, simply becuase we cant apply them to all members of
-- the category, specifically graphs of type EF a, it can only be applied to an adjmap. It is possible to
-- apply it to an EF a by first using gcoerce but it would be a lot nicer if we could actually apply
-- the functor straight to anything that satisifed the Graph constraint. I've tried to make this work
-- but it would require some type level programming stuff that I don't understand yet. The crux of the
-- problem seems to be that when defining graph functions (overlay, connect...) for EF the actual type
-- of the graph that these would be applied to is unknow at compile time (since it is any graph that
-- implements GraphCoerce). Due to time constraints I have opted to leave this as it is.

-- Ideally EF would look something like this:
-- data EF a where EFdata :: (Graph a, Graph b, Vertex b ~ [Vertex a], GraphCoerce b) => b -> EF a

------ The Ehrenfeucht-Fraisse Comonad ------
-- The comonad given here is actually a family of comonads. Each member of the family is assosiated
-- with an index k. The universe of the k indexed comonad has the invarient:
-- forall xs in (universe (EF A)). 1 <= length xs <= k.
-- For convinience we do not explicitally express the k parameter.

data EF a where EFdata :: (Graph a) => AdjacencyMap [Vertex a] -> EF a

instance (Graph a, Ord a, Ord (Vertex a)) => Graph (EF a) where
    type Vertex (EF a) = [Vertex a] -- Ideally this would be none empty lists of size k
    empty = EFdata AdjMap.empty
    vertex v = EFdata $ AdjMap.vertex v
    overlay (EFdata g1) (EFdata g2) = EFdata $ AdjMap.overlay g1 g2
    connect (EFdata g1) (EFdata g2) = EFdata $ AdjMap.connect g1 g2

instance (Graph a, Ord a, Ord (Vertex a)) => GraphCoerce (EF a) where
```

```
    gcoerce (EFdata g) = g
    gcoerceRev g       = EFdata g

-- inits for non-empty lists
-- inits [1,2,3] = [[1],[1,2],[1,2,3]]
ninits :: [a] -> [[a]]
ninits [] = []
ninits xs = ninits (init xs) ++ [xs]

-- exists k. forall Graph A,xs in (universe (EF A)). 1 <= length xs <= k
instance CComonad EF GraphMorphism where
    counit          = GM last
    extend (GM f)   = GM $ map f Prelude.. ninits


-- We automatically get the definition of the EF functor from the defn in Category.hs

deriving instance (Graph a, Ord a, Show a, Ord (Vertex a), Show (Vertex a)) => Show (EF a)

------ The Pebbling Comonad ------
-- The comonad given here is actually a family of comonads. Each member of the family is assosiated
-- with an index k. The universe of the k indexed comonad has the invarient:
-- forall xs in (universe (Pebble A)). max (peb-index xs) <= k && xs != []
-- For convinience we do not explicitally express the k parameter.

data Pebble a where Peb :: (Graph a) => AdjacencyMap [(Int, Vertex a)] -> Pebble a

instance (Graph a, Ord a, Ord (Vertex a)) => Graph (Pebble a) where
    type Vertex (Pebble a) = [(Int,Vertex a)]
    empty = Peb AdjMap.empty
    vertex v = Peb $ AdjMap.vertex v
    overlay (Peb g1) (Peb g2) = Peb $ AdjMap.overlay g1 g2
    connect (Peb g1) (Peb g2) = Peb $ AdjMap.connect g1 g2

instance (Graph a, Ord a, Ord (Vertex a)) => GraphCoerce (Pebble a) where
    gcoerce (Peb g) = g
    gcoerceRev g    = Peb g

-- exists k. forall Graph A, xs in (universe (Pebble A)). max (peb-index xs) <= k && xs != []
instance CComonad Pebble GraphMorphism where
    counit          = GM $ snd Prelude.. last
    extend (GM f)   = GM $ map (\xs -> (fst (last xs),f xs)) Prelude.. ninits

------ Useful functors ------

data Product a b where Prod :: (Graph a, Graph b) => AdjacencyMap (Vertex a, Vertex b) -> Product a b

instance (Graph a, Graph b, Ord (Vertex a), Ord (Vertex b)) => Graph (Product a b) where
    type Vertex (Product a b) = (Vertex a, Vertex b)
    empty = Prod AdjMap.empty
    vertex v = Prod $ AdjMap.vertex v
    overlay (Prod g1) (Prod g2) = Prod $ AdjMap.overlay g1 g2
    connect (Prod g1) (Prod g2) = Prod $ AdjMap.connect g1 g2

instance (Graph a, Graph b, Ord (Vertex a), Ord (Vertex b)) => GraphCoerce (Product a b) where
    gcoerce (Prod g) = g
    gcoerceRev g     = Prod g

instance CBifunctor Product GraphMorphism GraphMorphism GraphMorphism where
  bifuncMap (GM gm1) (GM gm2) = GM (\(x,y) -> (gm1 x, gm2 y))

product :: (Ord a, Ord b) => AdjacencyMap a -> AdjacencyMap b -> Product (AdjacencyMap a) (AdjacencyMap b)
product g1 g2 = Prod $ box g1 g2


data Coproduct a b where Coprod :: (Graph a, Graph b) => AdjacencyMap (Either (Vertex a) (Vertex b)) -> Coproduct a b

instance (Graph a, Graph b, Ord (Vertex a), Ord (Vertex b)) => Graph (Coproduct a b) where
    type Vertex (Coproduct a b) = Either (Vertex a) (Vertex b)
    empty = Coprod AdjMap.empty
```

```
    vertex v = Coprod $ AdjMap.vertex v
    overlay (Coprod g1) (Coprod g2) = Coprod $ AdjMap.overlay g1 g2
    connect (Coprod g1) (Coprod g2) = Coprod $ AdjMap.connect g1 g2

instance (Graph a, Graph b, Ord (Vertex a), Ord (Vertex b)) => GraphCoerce (Coproduct a b) where
    gcoerce (Coprod g) = g
    gcoerceRev g       = Coprod g

instance CBifunctor Coproduct GraphMorphism GraphMorphism GraphMorphism where
  bifuncMap (GM gm1) (GM gm2) = GM g
    where g (Left x)  = Left $ gm1 x
          g (Right x) = Right $ gm2 x

coproduct :: (Ord a, Ord b) => AdjacencyMap a -> AdjacencyMap b -> Coproduct (AdjacencyMap a) (AdjacencyMap b)
coproduct g1 g2 = Coprod $ AdjMap.overlay (gmap Left g1) (gmap Right g2)

-- Gives the equiliser graph and its the equaliser morphism.
-- G -> A --> B
-- /\ /\  -->
-- | /
-- Z
-- This finds all the vertacies in A for which all edges they are included in are preserved by f and g in B.
-- It then builds the subgraph of A, G that only includes these vertacies

-- Not really a true equaliser, for that we would need to introduce a new type, c, that contained only the members
-- of 'a' that were preserved by the homomorpism. We have no reasonable way of getting the action of the morpisms
-- elements in the universe that are not in edges without enumerating the type so its not really feasable to get
-- the true equiliser
getEqualiser :: (GraphCoerce a, GraphCoerce b, Ord (Vertex a), Eq (Vertex a), Eq (Vertex b)) =>
        a -> b -> GraphMorphism a b -> GraphMorphism a b -> a
getEqualiser g1 g2 (GM gm1) (GM gm2) = gcoerceRev (AdjMap.edges keptE)
    where vinE      = nub (concatMap (\(x,y) -> [x,y]) (edgeList adjg1))
          keptV     = map fst (intersect (map (\x -> (x,gm1 x)) vinE) (map (\x -> (x,gm2 x)) vinE))
          keptE     = filter (\(x,y)-> elem x keptV && elem y keptV) (edgeList adjg1)
          adjg1     = gcoerce g1

------ Useful functions ------

-- Apply a graph homomorphism to a graph
apply :: (Graph a, Graph b, GraphCoerce a, GraphCoerce b, Ord (Vertex a), Ord (Vertex b)) =>
         GraphMorphism a b -> a -> b
apply (GM gm) g = gcoerceRev (gmap gm (gcoerce g))


-- Check if each edge in g1 is mapped by gm to an edge in g2
checkMorphIsHomo :: (Graph a, Graph b, GraphCoerce a, GraphCoerce b, Eq (Vertex b)) =>
                    a -> b -> GraphMorphism a b -> Bool
checkMorphIsHomo g1 g2 (GM gm) = foldr (\e b -> elem e eG2 && b) True eMapped
    where eMapped = map (\(x,y) -> (gm x,gm y)) (edgeList (gcoerce g1))
          eG2     = edgeList (gcoerce g2)


-- Get all edges in g1 (and their mappings under gm) that are not mapped to an edge in g2 by gm
checkMorphIsHomoDebug :: (Graph a, Graph b, GraphCoerce a, GraphCoerce b, Eq (Vertex b))
                          => a -> b -> GraphMorphism a b -> [((Vertex a, Vertex a),(Vertex b, Vertex b))]
checkMorphIsHomoDebug g1 g2 (GM gm) = [(e1,e2)| (e1,e2) <- eMapped,not (elem e2 eG2)]
    where eMapped = map (\(x,y) -> ((x,y),(gm x,gm y))) (edgeList (gcoerce g1))
          eG2     = edgeList (gcoerce g2)

-- vertices of a graph
gvertices :: (GraphCoerce a, Eq (Vertex a)) => a -> [Vertex a]
gvertices = nub Prelude.. vertexList Prelude.. gcoerce

-- Given a list of values generate all possible plays of lenght k
plays :: Eq a => Int -> [a] -> [[a]]
plays k uni
    | length uni < k = (plays (k-1) uni) ++ (f (k-(length uni)) (permutations uni))
    | otherwise      = concatMap (lengthksublists uni) [1..k]
        where f 0 xs = xs
              f i xs = nub $ concatMap (\x -> concatMap (allinserts x) pf) uni
```

```
                        where pf = (f (i-1) xs)

-- Gives all different ways x can be inserted into ys
allinserts :: t -> [t] -> [[t]]
allinserts x []     = [[x]]
allinserts x (y:ys) = (x:y:ys) : map (y:) (allinserts x ys)

-- Get all length k lists that can be made from elements of xs (does not preserve order)
lengthksublists :: [a] -> Int -> [[a]]
lengthksublists xs 0 = [[]]
lengthksublists xs k = concatMap f (elemPairs xs)
    where f (x,ys) = map (x:) (lengthksublists ys (k-1))

--elemPairs [1,2,3] = [(1,[2,3]),(2,[1,3]),(3,[1,2])]
elemPairs :: [a] -> [(a, [a])]
elemPairs []     = []
elemPairs (x:xs) = (x,xs) : (map (\(y,ys) -> (y,x:ys)) (elemPairs xs))

-- The action of the natural transformation from ID to EFk
graphToEFk :: (GraphCoerce a, Eq (Vertex a), Ord (Vertex a)) => Int -> a -> EF a
graphToEFk k g = EFdata $ AdjMap.edges $
    concatMap (\p -> mapMaybe (\e -> f e p) edgesOfg) ps
    where edgesOfg  = edgeList (gcoerce g)
          ps        = plays k (gvertices g)
          f (a,b) p = maybePair (getPrefix p a, getPrefix p b)

maybePair :: (Maybe a, Maybe b) -> Maybe (a, b)
maybePair (Just a, Just b) = Just (a,b)
maybePair _ = Nothing

-- get prefix of play ending in y
getPrefix :: Eq t => [t] -> t -> Maybe [t]
getPrefix [] _ = Nothing
getPrefix (x:xs) y
            | x==y      = Just [x]
            | otherwise =  f (getPrefix xs y)
                where f (Just r) = Just (x:r)
                      f Nothing  = Nothing

-- graphToEFk usually gives a massive graph so this gives a subgraph of it
graphToLimEFk :: (GraphCoerce a, Eq (Vertex a), Ord (Vertex a)) => Int -> Int -> a -> EF a
graphToLimEFk lim k g = EFdata $ AdjMap.edges $
    concat $ take lim $ map (\p -> map (\(x,y) -> (f p x, f p y)) edgesOfg) ps
    where edgesOfg = edgeList (gcoerce g)
          ps       = plays k (gvertices g)
          f (x:xs) y  -- get prefix of play ending in y
            | x==y      = [x]
            | otherwise = x:(f xs y)

-- Check if efg is a valid EF k graph built from g
checkValidEFkGraph :: (Eq (Vertex g1), GraphCoerce g1, GraphCoerce g2, Vertex g2 ~ [Vertex g1]) =>
                      Int -> g1 -> g2 -> Bool
checkValidEFkGraph k g efg = foldr f True (edgeList (gcoerce efg))
    where gedges      = edgeList (gcoerce g)
          f (xs,ys) b = length xs <= k && length ys <= k
                        && elem (last xs,last ys) gedges
                        && (isPrefixOf xs ys || isPrefixOf ys xs)
                        && b

-- Check if pebg is a valid Peb k graph built from g
checkValidPebkGraph :: (Eq (Vertex g1), Graph g1, Graph g2, GraphCoerce g1, GraphCoerce g2,
                        Vertex g2 ~ [(Int,Vertex g1)]) => Int -> g1 -> g2 -> Bool
checkValidPebkGraph k g pebg = foldr f True (edgeList (gcoerce pebg))
    where gedges      = edgeList (gcoerce g)
          checkk xs   = foldr (\(x,y) b' -> b' && x <= k) True xs
          f (xs,ys) b = checkk xs && checkk ys
                        && elem (snd (last xs),snd (last ys)) gedges
                        && g1 xs ys
                        && b
          g1 xs ys
```

```
            | isPrefixOf xs ys = h xs ys
            | isPrefixOf ys xs = h ys xs
            | otherwise        = False
                where h xs' ys' = foldr (\(x,_) b' -> (lastx /= x) && b') True (drop (length xs') ys')
                        where lastx     = fst (last xs')

-- Given a duplicator strategy builds the graph morphism f: EFk(A) -> B
buildMorphFromStrat :: (Show (Vertex a), Show (Vertex b), GraphCoerce a, GraphCoerce b, Ord a, Ord (Vertex a),
                        Ord (Vertex b), Eq (Vertex a), Eq (Vertex b)) =>
                        (Int -> a -> b -> [Vertex a] -> [Vertex b]) -> Int -> a -> b ->
                        CoKleisli EF GraphMorphism a b
buildMorphFromStrat strat k glin1 glin2 = CoKleisli $ GM (last Prelude.. strat k glin1 glin2)


---------- Proof checkers ----------

-- Checks for equailty in the existential positive fragment with quantifier rank k
eqQRankKEPfrag :: (Eq a, Eq b, Ord a, Ord b, Ord (Vertex a), Ord (Vertex b), GraphCoerce a, GraphCoerce b)
                    => Int -> CoKleisli EF GraphMorphism a b -> CoKleisli EF GraphMorphism b a -> a -> b -> Bool
eqQRankKEPfrag k (CoKleisli h1) (CoKleisli h2) g1 g2 =
                    (AdjMap.isSubgraphOf (gcoerce (apply h1 (graphToEFk k g1))) (gcoerce g2)) &&
                    (AdjMap.isSubgraphOf (gcoerce (apply h2 (graphToEFk k g2))) (gcoerce g1))

-- Checks for equailty in the fragment quantifier rank k with counting quantifiers (i.e. checks for isomorphism)
eqQRankKCounting :: (Eq a, Eq b, Ord a, Ord b, Ord (Vertex a), Ord (Vertex b), GraphCoerce a, GraphCoerce b)
                    => Int -> CoKleisli EF GraphMorphism a b -> CoKleisli EF GraphMorphism b a -> a -> b -> Bool
eqQRankKCounting k (CoKleisli h1) (CoKleisli h2) g1 g2 =
                    (apply h1 (graphToEFk k g1) == g2) && (apply h2 (graphToEFk k g2) == g1)
```

A.3 The Examples

```
{-# LANGUAGE ConstraintKinds        #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE UndecidableInstances   #-}
{-# LANGUAGE StandaloneDeriving     #-}



module AlgGraphsExamples where

import Algebra.Graph.Class
import Algebra.Graph.AdjacencyMap as AdjMap
import qualified Algebra.Graph.NonEmpty.AdjacencyMap as NonEmptyAdjMap (fromNonEmpty, overlay, edge, edgeList)
import Algebra.Graph.AdjacencyMap.Algorithm


import GHC.Exts (Constraint)
import Data.List
import Data.Maybe
import Data.Coerce
import Data.Tree
import Data.Map as Map (Map, fromList, insert, toList, empty, lookup, elems)
import Data.Set as Set (Set, fromList)
import System.Random hiding (split)
import Debug.Trace
import Category
import AlgGraphsCat
import DrawGraphs

------------------ Proof of concept example with two equivilent graphs  --------------------
-- With EF comonad I can only prove them equivilent in fragment of logic with quantifier rank k.
-- To do this we give a coklislie morphism that takes the EK graph with play length k to graph b

-- Not acctually random, but generates arbitrary graphs given a seed
```

```haskell
randomGraph :: Int -> Int -> AdjacencyMap Int
randomGraph seed n = fromAdjacencySets $ f verts nums
    where nums :: [Int]
          nums       = take (n*n) (randomRs (1, 2) (mkStdGen seed))
          verts      = [1..n]
          f [] r     = []
          f (v:vs) r = (v, Set.fromList [y |(x,y)<- zip rands verts, x==1]) : (f vs newr)
                       where (rands,newr) = splitAt n r

-- randomGraphList size num = map (\x -> randomGraph (mkStdGen x) size) [1..num]

liftMapToEFMorph ::(Graph a, Graph b, Ord (Vertex a), Ord a) =>
                   (Vertex a -> Vertex b) -> CoKleisli EF GraphMorphism a b
liftMapToEFMorph f = CoKleisli $ GM f Category.. counit

-- Proof that two isomorphic graphs are equal in the fragment quantifier rank k with counting quantifiers
-- Provide any valid iso (A->B, B->A)
proof1 :: Int -> Int -> Int -> (Int -> Int, Int -> Int) -> Bool
proof1 seed size k iso = eqQRankKCounting k g1tog2 g2tog1 graph1 graph2
    where graph1 = randomGraph seed size
          graph2 = gmap (fst iso) graph1
          g1tog2 = liftMapToEFMorph (fst iso)
          g2tog1 = liftMapToEFMorph (snd iso)

ex1 = proof1 45 3 4 ((\x -> x*2), (\x -> x `div` 2))

-- Proof that two isomorphic graphs are equal in the exisential positvie fragment with quantifier rank k
proof2 seed size k iso = eqQRankKEPfrag k g1tog2 g2tog1 graph1 graph2
    where graph1 = randomGraph seed size
          graph2 = gmap (fst iso) graph1
          g1tog2 = liftMapToEFMorph (fst iso)
          g2tog1 = liftMapToEFMorph (snd iso)

ex2 = proof2 45 3 4 ((\x -> x*2), (\x -> x `div` 2))
------------------ Example with linear orderings  --------------------
-- Given a list representing a linear ordering it returns the equivilent
-- graph with the edge relation interpreted as the < relation
linToGraph :: Ord a => [a] -> AdjacencyMap a
linToGraph xs = AdjMap.edges (f xs)
  where f [] = []
        f (x:xs) = map (\y -> (x,y)) xs ++ f xs

-- Counts occurrences of x in a list
count :: Eq a => a -> [a] -> Int
count x = foldr (\y xs-> if x==y then xs + 1 else xs) 0

-- converts a graph to a linear order
-- Pre: Graph is a representation of a lin order
graphToLin :: (GraphCoerce a, Eq (Vertex a)) => a -> [Vertex a]
graphToLin g = reverse $ map fst $ sortBy (\(_,a) (_,b) -> compare a b) $ map (\x -> (x,count x e)) (gvertices g)
    where e = map fst (edgeList (gcoerce g))

-- Splits the list into two parts on x with the middle element repeting
-- Pre: elem x xs
split :: Eq a => a -> [a] -> ([a],[a])
split x xs = f [] xs
    where f xs (y:ys) = if x == y then (reverse (x:xs), x:ys) else f (y:xs) ys

-- Splits the list into two parts at index i with middle element repeting
splitAtD :: Int -> [a] -> ([a],[a])
splitAtD i xs = (\(x,y) -> (x++[head y],y)) $ splitAt i xs

-- Get position of y in xs
-- Pre: y is in xs
index y xs = f xs 0
    where f (x:xs) i
            | x == y    = i
            | otherwise = f xs (i+1)

-- Like lookup but crashes if x isnt in the list
```

```haskell
-- Pre: x is in the iso
getIso :: Eq a => [(a,b)] -> a -> b
getIso ((a,b):iso) x
    | x == a     = b
    | otherwise  = getIso iso x

-- Check if x occurs as first elem in any of the pairs
inIso :: Eq a => a -> [(a,b)] -> Bool
inIso x []    = False
inIso x ((a,b):ps)
    | x==a      = True
    | otherwise = inIso x ps

both :: (a -> b) -> (a, a) -> (b, b)
both f (x,y) = (f x, f y)

-- Take two linear orders as graphs, and a spoilers play, return the duplicators play
-- Pre: Both the linear orders are at least length 2^k.
linStrategy :: (Graph a, Graph b, GraphCoerce a, GraphCoerce b, Eq (Vertex a), Eq (Vertex b))
                        => Int -> a -> b -> [Vertex a] -> [Vertex b]
linStrategy k glin1 glin2 spoil = map (getIso iso) spoil
    where lin1 = graphToLin glin1
          lin2 = graphToLin glin2
          iso  = linRec k lin1 lin2 spoil []

-- Recusivly implement the strategy

-- Keep the iso in the argument and pass it around, if somethings already in the iso then we
-- can continue on to the next spoiler play. If not then for the first two cases we want to
-- zip together the appropriate parts of the list, add them to the iso and if the spoiler plays
-- something not in the iso, call linRec on the remaining bit. It may seem strange that there will
-- be points on recursive calls of linRec that the spoiler can play outside lin1 but in that
-- case it should be in iso.
linRec :: (Eq a, Eq b) => Int -> [a] -> [b] -> [a] -> [(a,b)] -> [(a,b)]
linRec _ _ _ [] iso = iso
linRec 0 _ _ _  iso = iso
linRec k lin1 lin2 (a:ps) iso
    | inIso a iso             = linRec (k-1) lin1 lin2 ps iso
    | length lin1LT < 2^(k-1) = linRec (k-1) lin1GT lin2GTc1 ps (iso ++ zip lin1LT lin2LTc1)
    | length lin1GT < 2^(k-1) = linRec (k-1) lin1LT lin2LTc2 ps (iso ++ zip lin1GT lin2GTc2)
    | otherwise               = lemma (k-1) (lin1LT,lin1GT) (lin2LTc3,lin2GTc3) ps ((a,bc3):iso)
      where (lin1LT,lin1GT)     = split a lin1
            (lin2LTc1,lin2GTc1) = splitAtD (index a lin1) lin2
            (lin2GTc2,lin2LTc2) = both reverse (splitAtD (length lin1 - index a lin1) (reverse lin2))
            (lin2LTc3,lin2GTc3) = splitAtD (2^(k-1)) lin2
            bc3                 = lin2 !! (2^(k-1))

-- The lemma from the proof
lemma :: (Eq a, Eq b) => Int -> ([a], [a]) -> ([b], [b]) -> [a] -> [(a, b)] -> [(a, b)]
lemma k (lin11,lin12) (lin21,lin22) ps iso = nub $ linRec k lin11 lin21 p1 iso ++ linRec k lin12 lin22 p2 iso
    where (p1,p2) = partition (flip elem lin11) ps


-- checkEFkMorph :: (Ord a, Ord b) => Int -> AdjacencyMap a -> AdjacencyMap b -> AdjacencyMap b
-- checkEFkMorph k glin1 glin2 = apply ((buildMorphFromStrat linStrategy) k glin1 glin2) eflin1
--     where eflin1 = graphToEFk k glin1


--res2 = checkEFkMorph 2 lin6 lin7

--res3 = checkMorphIsHomo (graphToEFk 2 lin4) lin5 (buildLinMorph 2 lin4 lin5)
--res4 = checkMorphIsHomo (graphToEFk 4 lin1) lin2 (buildLinMorph 4 lin1 lin2)


-- Would be good to abstract this to some "checkStrat" function but due to unification problems
-- its not really possible (applying buildMorph g1 g2, then buildMorph g2 g1 is not good for ghc)

-- Pre: the two linear orders are at least lenght 2^k
proof3 k lin1 lin2 = eqQRankKEPfrag k gm1 gm2 g1 g2
    where gm1 = buildMorphFromStrat linStrategy k g1 g2
```

```
          gm2 = buildMorphFromStrat linStrategy k g2 g1
          g1 = linToGraph lin1
          g2 = linToGraph lin2

ex3 = proof3 2 [1..5] [7..11]
ex4 = proof3 3 [1..9] [10..18]
-- Takes ages
ex5 = proof3 4 [1..17] [3..22]

------------------ Two colourability/Even/Connectivity  --------------------
-- We can get two undriected graphs, one thats a cycle of size (2^k)+1, and one thats made up
-- of two cycles, both of size 2^k. Then for the k round EF game they are equal. Since the single
-- cycle graph is 2 colourable, odd and connected and the two cycle graph is even, unconnected
-- and not 2-colourable showing that they're equal with eqQRankKEPfrag k constitues a proof that
-- they're equal in the equivilent logic.

-- Function to generate the two graphs
generateCycleG :: Int -> (AdjacencyMap Int, AdjacencyMap Int)
generateCycleG k = (symmetricClosure (AdjMap.circuit [1..2^k+1]),
        symmetricClosure (AdjMap.overlay (AdjMap.circuit [1..2^k]) (AdjMap.circuit [2^k+1..2^(k+1)])))

-- Duplicators strategy is to ensure that the distance between any two of her pebbles is either
-- equal to the distance between the corrosponding pair of the spoilers pebbles, or, it is greater
-- than 2^k-r. If the two spoilers pebbels are on the same graph then it needs to be equal, otherwise
-- greater than 2^k
cycStrategy :: (Show (Vertex a), Show (Vertex b), Graph a, Graph b, GraphCoerce a, GraphCoerce b,
                Ord (Vertex a), Ord (Vertex b)) => Int -> a -> b -> [Vertex a] -> [Vertex b]
cycStrategy k g1 g2 spoil = f spoil [] [] 1
    where f []     ss ds r = ds
          f (x:xs) ss ds r = f xs (ss++[x]) (ds++[g (findEqualPlacing x (gcoerce g1) (gcoerce g2) ss ds)]) (r+1)
            where g (Just y) = y
                  g Nothing  = findFarPlacing k r x (gcoerce g1) (gcoerce g2) ss ds

-- Remove all elems of xs from ys
removeL :: Eq a => [a] -> [a] -> [a]
removeL xs ys = filter (\y -> not (elem y xs)) ys

-- Find a placing in g2 of distance 2^(k-r) away from all others, if it is the last node to be places,
-- make sure it respects adjancecy since it will be of dist 1 away
findFarPlacing :: (Eq a, Ord a, Ord b) => Int -> Int -> a -> AdjacencyMap a -> AdjacencyMap b -> [a] -> [b] -> b
findFarPlacing k r n g1 g2 spoil dup = if d==1 && t then head (intersect neighbours2 placings) else head placings
    where placings    = foldr (\x xs -> intersect (getGTIAway x d g2) xs) (vertexList g2) dup
          d           = 2^(k-r)
          t           = elem (last spoil) neighbours1
          neighbours1 = getIAway n 1 g1
          neighbours2 = getIAway (last dup) 1 g2

-- get closer nodes then take them away from whole graph
-- Get all nodes of distance i or larger from n in g
getGTIAway :: (Eq a, Ord a) => a -> Int -> AdjacencyMap a -> [a]
getGTIAway n i g = removeL (f 0 [] [n]) (vertexList g)
    where f d visited []    = visited
          f d visited nodes
            | d>=i      = visited
            | otherwise = f (d+1) newV newN
                where newN = concatMap (\x -> removeL newV (fromJust (Map.lookup x adjMap))) nodes
                      newV  = visited ++ nodes
          adjMap = Map.fromList (adjacencyList g )

-- Find placing of node on g2 s.t. it preserves the none-infinite distances to the nodes in g1
-- If this is impossible return Nothing.
-- Dup and spoil should be the same length
-- Go through nodes close to n in other graph and see if there are any positions you can place the new one that
-- Keep the distances
-- If the distances of all spoilers plays are Nothing then return Nothing
findEqualPlacing :: (Show a, Show b, Eq a, Ord a, Ord b) =>
                    a -> AdjacencyMap a -> AdjacencyMap b -> [a] -> [b] -> Maybe b
findEqualPlacing n g1 g2 spoil dup = if incomparableS then Nothing
                                                      else listToMaybe $ removeL dup $ foldr f g1Vlist (zip spoil dup)
    where spoilDists   = getCycDistances n g1 spoil
```

```
          f (x,y) xs   = intersect (g y (fromJust (Map.lookup x spoilDists))) xs
          g y (Just d)   = getIAway y d g2
          g y Nothing    = g1Vlist
          incomparableS  = foldr (\x xs -> (x==Nothing) && xs) True (Map.elems spoilDists)
          g1Vlist      = vertexList g2

-- Get all nodes i away from n on a cycle graph
getIAway :: (Eq a, Ord a) => a -> Int -> AdjacencyMap a -> [a]
getIAway n i g = nub (f 0 [] [n])
    where f d visited []        = []
          f d visited nodes
            | d==i      = nodes
            | otherwise = f (d+1) newV newN
                where newN = concatMap (\x -> removeL newV (fromJust (Map.lookup x adjMap))) nodes
                      newV  = visited ++ nodes
          adjMap = Map.fromList (adjacencyList g )

-- Given a node and a graph find the distance from it to each other node in spoil (Nothing if infinite)
-- Only works graphs that are a collection of cycles
-- Dists should be a map
getCycDistances :: (Show a, Eq a, Ord a) => a -> AdjacencyMap a -> [a] -> Map a (Maybe Int)
getCycDistances n g spoil = foldr (\x xs-> Map.insert x (Map.lookup x listOfDists) xs) Map.empty spoil
    where listOfDists                = f 0 [] [n] Map.empty
          f d visited [] dists       = dists
          f d visited nodes dists = f (d+1) newV newN newD
                where newN = concatMap (\x -> removeL newV (fromJust (Map.lookup x adjMap))) nodes
                      newD  = foldr (\x xs -> Map.insert x d xs) dists nodes
                      newV  = visited ++ nodes
          adjMap = Map.fromList (adjacencyList g)


-- getEFkMorphCyc :: (Show a, Show b, Ord a, Ord b) => Int -> AdjacencyMap a -> AdjacencyMap b -> AdjacencyMap b
-- getEFkMorphCyc k glin1 glin2 = apply (buildMorphFromStrat cycStrategy k glin1 glin2) eflin1
--      where eflin1 = graphToEFk k glin1


-- res7 = getEFkMorphCyc k g1 g2

-- res8 = checkMorphIsHomo (graphToEFk k g1) g2 (buildCycleMorph k g1 g2)

-- res11 :: EF (AdjacencyMap Int)
-- res11 = apply (GM ( k g1 g2)) (graphToEFk k g1)
res13 = checkMorphIsHomoDebug (graphToEFk k g1) g2 gm
    where (g2,g1) = generateCycleG k
          k = 5
          (CoKleisli gm) = buildMorphFromStrat cycStrategy k g1 g2

-- Takes ages if k is 4, fine for 2 and 3
proof5 k = eqQRankKEPfrag k gm1 gm2 g1 g2
    where gm1 = buildMorphFromStrat cycStrategy k g1 g2
          gm2 = buildMorphFromStrat cycStrategy k g2 g1
          (g1,g2) = generateCycleG k


------------------ Cycle and line graph (tree is not expressable) ------------------


------------------ Hamiltonian --------------------

-- This will constitue a proof that a hamiltonian graph is not axiomatisable with a finite number
-- of variables in FO logic

-- Two graphs, the first has a hamiltonian cycle, the second does not
generateHamiltonianG :: Int -> (AdjacencyMap Int, AdjacencyMap Int)
generateHamiltonianG k = (symmetricClosure (AdjMap.biclique [1..k] [k+1..2*k]),
                          symmetricClosure (AdjMap.biclique [1..k] [k+1..(2*k+1)]))

-- Strategy is to pick any elem on the same side not already in iso, since there are always k or more on each side
-- and the nodes on a given side are indistingushable this is a winning strategy.
-- Only works on hamitolians generated with generateHamiltonianG
```

```haskell
--Basically choose an unused elem in the same side of the graph
hamStrategy :: Int -> [(Int,Int)] -> [(Int,Int)]
hamStrategy k spoil = hamStratRec k spoil []

-- find elems on one side of the graph that arnt currently in the iso, need to keep track of moved pebbles
unusedKLT :: Int -> [(Int,Int)] -> Int
unusedKLT k revdup = head (dropWhile (\x -> not (elem x usedNodes)) [1..k])
    where usedNodes = [a | (i,a) <- currentPos (reverse revdup),i<=k]

unusedKGT :: Int -> [(Int,Int)] -> Int
unusedKGT k revdup = head (dropWhile (\x -> not (elem x usedNodes)) [k+1..2*k+1])
    where usedNodes = [a | (i,a) <- currentPos (reverse revdup),i>k]

-- Given a play get the end position of all the pebbles (i.e. remove duplicate occurences of peb positions)
currentPos :: [(Int,a)] -> [(Int,a)]
currentPos ps = f ps Map.empty
    where f [] curpos         = Map.toList curpos
          f ((i,x):xs) curpos = f xs (Map.insert i x curpos)

-- revdup is the reverse of the duplicators play, i.e. if (1,a) occours before (1,b) in the list then
-- pebble 1 is on a
hamStratRec :: Int -> [(Int,Int)] -> [(Int,Int)] -> [(Int,Int)]
hamStratRec k [] revdup = reverse revdup
hamStratRec k ((pi,n):xs) revdup
    | n <= k     = hamStratRec k xs ((pi,unusedKLT k revdup):revdup)
    | otherwise  = hamStratRec k xs ((pi,unusedKGT k revdup):revdup)


-- Haskell needs the type of counit explicitally to work out which comonad it should use the counit for
hamMorphs :: (Graph b, Graph a, Vertex a ~ Int, Vertex b ~ Int, Ord a, Ord b) =>
    Int -> (CoKleisli Pebble GraphMorphism a b,
            CoKleisli Pebble GraphMorphism b a)
hamMorphs k = (CoKleisli $ f Category.. GM (hamStrategy k),
               CoKleisli $ f Category.. GM (hamStrategy k))
    where f :: (Graph g, Ord g, Ord (Vertex g)) => GraphMorphism (Pebble g) g
          f = counit

------------------ Tree depth experiments  --------------------

data Gaifman a where Gaifman :: (Graph a, Ord a, Ord (Vertex a)) => AdjacencyMap (Vertex a) -> Gaifman a

-- The gaifman graph is undirected and irreflexive (no self edges)
-- Technically I think the Gaifman graph should include all elems of the universe, i.e. the type but this
-- would not be practical
getGaifmanGraph :: Ord a => AdjacencyMap a -> Gaifman (AdjacencyMap a)
getGaifmanGraph g = Gaifman $ AdjMap.edges (nub (concat newedges))
    where newedges = [[(v1,v2),(v2,v1)] |(v1,v2) <- (edgeList g), v1 /= v2]
-- Get connected componants of a Gaifman graph
getCC :: Gaifman (AdjacencyMap a) -> [Gaifman (AdjacencyMap a)]
getCC (Gaifman g) = map (Gaifman Prelude.. NonEmptyAdjMap.fromNonEmpty) $ vertexList $ scc g

-- Tree depth decomposition of the Gaifman graph


-- This applies the following method for treedepth decomposition to each connected componant
-- of gg.
-- A treedepth decomposition of a connected graph G=(V,E) is a rooted tree T=(V,ET) that
-- can be obtained in the following recursive procedure. If G has one vertex, then T=G.
-- Otherwise pick a vertex v in V as the root of T, build a treedepth decomposition of each
-- connected component of G-v and add each of these decompositions to T by making its root
-- adjacent to v
gaifmanTDD :: Gaifman (AdjacencyMap a) -> [Tree a]
gaifmanTDD gg = map f (getCC gg)
    where f (Gaifman g)
            | vertexCount g == 1 = Node (head (vertexList g)) []
            | otherwise          = Node v (map f cc)
              where v  = head (vertexList g)
                    cc = getCC (Gaifman (removeVertex v g))
```

```haskell
deriving instance (Graph a, Ord a, Ord (Vertex a), Show (Vertex a)) => Show (Gaifman a)

------------------ Coalgebra to forest cover  --------------------
-- We assume that the coalgebra takes elems of the universe not in any relation (edge) to their singleton list
-- We then do not include them in the forest cover, since this would be impracticle.

-- Send each element to the list of its predecessors in the tree
forestToCoalg :: (Eq a, Ord a) => Forest a -> GraphMorphism (AdjacencyMap a) (EF (AdjacencyMap a))
forestToCoalg forest = (GM f)
    where f a = head $ mapMaybe (findNodePreds a) forest

-- Pre: The node a occurs only once in the tree
-- Find the predecsors of a node
findNodePreds :: Eq a => a -> Tree a -> Maybe [a]
findNodePreds a t
    | rootLabel t == a  = Just [a]
    | subForest t == [] = Nothing
    | otherwise         = if null levelBelow then Nothing else Just $ (rootLabel t) : (head levelBelow)
      where levelBelow  = mapMaybe (findNodePreds a) (subForest t)

-- We also need the domain graph of the coalg to work out which elems of the universe are in edges so we can
-- apply the coalg to them to get the forest
coalgToForest :: (GraphCoerce a, Eq (Vertex a), Ord (Vertex a)) => GraphMorphism a (EF a) -> a -> Forest (Vertex a)
coalgToForest (GM f) g = foldr addToForest [] (foldr h [] (vertexList (gcoerce g)))
    where h a l = updateList (f a) l
          updateList as []     = [as]
          updateList as (xs:xss)
            | isPrefixOf as xs = xs:xss
            | isPrefixOf xs as = as:xss
            | otherwise        = xs:(updateList as xss)
          buildForest p forest = forest

-- Doesnt work for empty lists since you cant have an empty tree
listToTree :: [a] -> Tree a
listToTree [x]     = Node x []
listToTree (x:xs)  = Node x [listToTree xs]

-- start with line tree, with each new tree its prefix closed so it needs to start at the root, follow down from root
-- until we have to branch off, then add on the branch
-- if the roots dont match then can be part of same play so must be seperate forest

addToForest :: Eq a => [a] -> Forest a -> Forest a
addToForest [] []    = []
addToForest p []     = [listToTree p]
addToForest p (t:ts) = if rootLabel t == head p then nt:ts else t:addToForest p ts
    where nt = mergeWithTree t p

-- Pre: x == p, (x=rootLable tx)
mergeWithTree :: Eq a => Tree a -> [a] -> Tree a
mergeWithTree tx [p]            = tx
mergeWithTree (Node x xs) (p:ps) = f (checkNextLevel xs (head ps))
    where f (Just t, ts) = Node x ((mergeWithTree t ps):ts)
          f (Nothing,ts) = Node x ((listToTree ps):ts)

checkNextLevel :: Eq a => [Tree a] -> a -> (Maybe (Tree a), [Tree a])
checkNextLevel [] a     = (Nothing,[])
checkNextLevel (x:xs) a = if rootLabel x == a then (Just x,ts) else (Nothing,x:ts)
    where (t,ts) = checkNextLevel xs a

testCoalgForest g = coalgToForest (forestToCoalg gg) g == gg
    where gg = gaifmanTDD (getGaifmanGraph g)

getTreeDepthOfDecomp :: [Tree a] -> Int
getTreeDepthOfDecomp f = maximum (map (foldTree (\_ xs -> if null xs then 1 else 1 + maximum xs)) f)

-- Checks if a graph and its assosiated object form a valid EFk coalgebra
checkValidCoalg :: (GraphCoerce a, Eq (Vertex a), Ord (Vertex a)) => GraphMorphism a (EF a) -> a -> Bool
checkValidCoalg gm g = checkValidTDD g (coalgToForest gm g)
```

```haskell
-- Checks if a forest is a valid forest
checkValidTDD :: (GraphCoerce a, Eq (Vertex a), Ord (Vertex a)) => a -> Forest (Vertex a) -> Bool
checkValidTDD g forest = foldr (\(x,y) xs -> xs && or (map (checkAorD x y) forest)) True es
    where es = edgeList (gcoerce g)

-- returns true if x is a decendent or ancestor of y
checkAorD :: (Eq a) => a -> a -> Tree a -> Bool
checkAorD x y (Node l ts)
    | x == l    = foldr (\z zs -> checkD y z || zs) False ts
    | y == l    = foldr (\z zs -> checkD x z || zs) False ts
    | otherwise = foldr (\z zs -> checkAorD x y z || zs) False ts

-- returns true if x is in the tree
checkD :: (Eq a) => a -> Tree a -> Bool
checkD x t = foldTree (\y ys -> if y == x then True else or ys) t
```