Implementing Game Comonads in Finite Model Theory using Dependent Types in Idris

Submitted by: Robert Kirk Supervised By: Samson Abramsky

Honour School of Mathematics and Computer Science - Part C

Somerville College, University of Oxford

May 16, 2018

Abstract

This report comprises an exposition of games in finite model theory, a comonadic formulation of such games, and a description of an implementation of this comonadic formulation in Idris, a language with dependent types. Games play a large part in the study of finite model theory, where certain bound on the games (such as the number of pebbles, or the number of rounds) can be used to model limited access to the structures the games are played on. Comonads are a construct of category theory and arise often in categorical semantics of computation as a way of modelling context dependent computations or functions. It was shown in [1] that we can model the playing of one of these games, the pebble game, using a comonad over the category of logical structures, and that this formulation captures entirely the nature of these games. Building on these results, and others which provide comonadic formulations for other games, I provide an implementation of the various category theory and finite model theory structures required for these comonads, and an implementation of the comonads themselves. This implementation is achieved in Idris, a language with dependent types, and I use this as an opportunity to demonstrate the power of Idris in expressing these kinds of concepts correctly and completely.

Acknowledgements

I'd like to thank my supervisor Samson Abramsky for his advice and guidance during this project, and for opening me up to a whole new world of type and category theory which I wouldn't have encountered otherwise.

I'm thankful to my personal tutor Quentin Miller for his support throughout my four years at Oxford; I couldn't have asked for a better personal tutor, and he's helped me develop immensely.

Finally, I'd like to thank my family and especially my girlfriend for their constant support and love, without which I couldn't have completed this project.

Contents

1. Introduction

Category Theory and Finite Model Theory are two distinct areas of Mathematics and Computer Science which haven't traditionally interacted or shared ideas. The aim of this report is two-fold: First, to explain and explore the paper by Samson Abramsky, Anuj Dawar and Pengming Wang in [1], and other results by Abramsky, where they use the concept of an indexed comonad from category theory to encapsulate the existential pebble game and the Ehrenfeucht-Fraïssé game from finite model theory. This enables us to give a comonadic and hence category theoretic characterisation of some central concepts in finite model theory. Secondly, I aim to give an exposition of Idris - the dependently typed functional programming language developed by Edwin Brady at University of St Andrews [4] - and how it can be used to implement many concepts from category theory and finite model theory, and used as a proof assistant to demonstrate correctness of arguments concerning the pebble game on certain structures.

In Chapter 2, I'll introduce the necessary background in category theory and finite model theory, and then using this knowledge I'll introduce the main theoretical objects of this report, the pebble comonad and the EF comonad. I'll go on to introduce Idris more fully, and explore the various features I will be using during the course of this report. In Chapter 3 I'll explain the implementation of the two game comonads in Idris, detailing how various parts of Idris interact with each other and explaining the proofs used in detailing the correctness of the implementation within Idris. In Chapter 4 I'll explain the variety of examples I have developed which use the pebble comonad, including basic examples of proofs which could be found in Leonid Libkin's book *Elements of Finite Model Theory* [6]. Chapter 5 will conclude, and explore some possible areas where this work could be build upon and expanded.

Notation and required Background

I will be assuming a certain amount of knowledge in category theory and finite model theory, and a basic background in functional programming. In terms of category theory I'm assuming knowledge which would be taught in a first basic course (e.g. [10]): the definitions of categories, morphisms, functors, natural transformations, adjunctions and dualising, and commuting diagrams. Anything more advanced than that (i.e. comonads and coalgebras) will be explained when the time comes. In terms of finite model theory, a background in logic would be useful, enough to understand concepts such as: signatures, models, relations, sentences, fragments of first order logic and satisfaction, all of which could be found for example in [6]. I will explain all material above this level.

In terms of understanding Idris, a background in functional programming would be useful, especially Haskell [9], as they share similarities in syntax and semantics. A

knowledge of various concepts in type theory such as dependent types and Sigma types, as well as experience with a proof language, may be useful in grasping the finer points of how Idris's features work, but they will all be explained in the relevant sections. I used the Idris tutorial to find most of the information I used, which can be found at http://docs.idris-lang.org/en/latest/, and is a slimmed down version of the book *Type Driven Development with Idris* by Edwin Brady [5].

Concerning notation, all notation will be introduced as it is used, and it can be assumed that using the same symbol as when the concept is introduced will normally stand for an arbitrary object of that concept, when the context is clear. For example when I introduce monads I use M to stand for the functor of the monad, so would normally refer to general monads throughout the paper as M.

2. Theoretical Background in Category Theory, Finite Model Theory and Idris

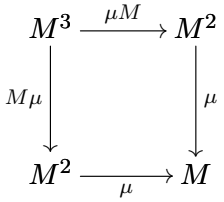
2.1. Monads and Comonads in Category Theory

Category theory is the study of categories and their related concepts. It's a way of abstracting the study of specific structural properties to more general settings. We can then use such results in a wide variety of mathematical and computer science disciplines. I'll introduce the concept of a monad here, as it often arises more naturally than a comonad. For a category $\mathcal{C}$ we define a monad to be a triple (M, η, μ) with:

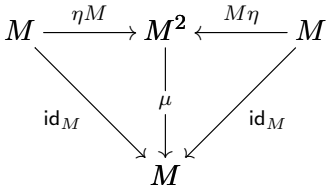
- $M : \mathcal{C} \to \mathcal{C}$ a functor, and hence an endofunctor
- $\eta : \mathsf{id} \Rightarrow M$ a natural transformation, called the unit
- $\mu : M^2 \Rightarrow M$ also a natural transformation, called the multiplication

which must satisfy the following equations

Associativity



Identity



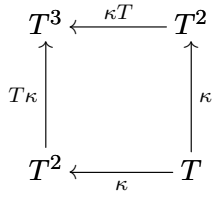
A monad is often a way of adding additional structure to a value, and they're the basis of computation with side effects in functional programming languages such as Haskell, where the monad carries any extra state required. A monad can also be viewed as a monoid (a group without inverses) in the category of endofunctors on the base category C, where the identity and associativity axioms state exactly the identity and associativity of this monoid.

In category theory, there is always the notion of dualising. Given a concept for a category $\mathcal{C}$, we can obtain the dual concept by considering the original idea applied to the opposite category $\mathcal{C}^{op}$ - the category with all morphisms reversed - and then converting back to the original category $\mathcal{C}$. The dual notion of a monad is that of a comonad, which similarly to monads can be defined in a number of different ways. I will give the standard mathematical definition here, and then demonstrate a different formulation. Thus, a comonad for a category $\mathcal{C}$ consists of:

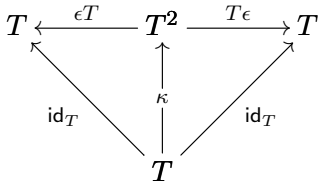
- $T : \mathcal{C} \to \mathcal{C}$ a functor, and hence an endofunctor
- $\varepsilon : T \Rightarrow \mathsf{id}$ a natural transformation, called the counit
- $\kappa : T \Rightarrow T^2$ also a natural transformation, called the comultiplication

which must satisfy the following equations

Coassociativity



Identity



Note these are very similar to the monad definitions but with the arrows reversed, as the comonad is simply a monad in the opposite category. While technically a comonad is a triple I will refer to a comonad just as T when the multiplication and comultiplication are obvious.

Just as a monad could be seen as a way of doing computations with extra effects which the monad carries, a comonad can be seen as a way of giving context to a computation or function, such as extra state or definitions, which the function may implicitly require.

There is another formulation of a comonad, called the Kleisli form. In this definition, we required a map T on the objects of the category, arrows $\varepsilon_{\mathcal{A}} : T \to \mathsf{id}$ for each object $\mathcal{A}$ of the category, and an operation $(\cdot)^* : (TA \to B) \to (TA \to TB)$ which we call the Kleisli coextension. These maps must obey the following properties:

$$\varepsilon^*_{\mathcal{A}} = \mathsf{id}_{TA} \qquad \varepsilon_{\mathcal{A}} \circ f^* = f \qquad (g \circ f^*)^* = g^* \circ f^*$$

With these definitions, we can get back to our original definitions as follows: We extend T to a functor by $Tf = (f \circ \varepsilon_{\mathcal{A}})^*$, and we define the comultiplication $\kappa : T \Rightarrow T^2$ by

$\kappa_A = (\mathsf{id}_{TA})^*$, then we have $(T,\ \epsilon,\ \kappa)$ as normal, and the equations hold by the conditions on the Kleisli form. Conversely, given the standard triple $(T,\ \epsilon,\ \kappa)$, we can define the extension as $f^* = Gf \circ \kappa_A$.

We can expand these definitions to introduce the concept of an indexed comonad $\mathbb{T}$. This is effectively a collection of comonads $\{T_i,\ \varepsilon_i,\ \kappa_i\}_{i \in N}$ for some set N. Equivalently, we can view it as a triple of functions:

$\mathbb{T} : N \to \mathcal{C} \to \mathcal{C}$
$\varepsilon : (i\ of\ type\ N) \to T_i \Rightarrow id$
$\kappa : (i\ of\ type\ N) \to T_i \Rightarrow T_i^2$

We can also write these comonads in Kleisli form as above. A priori there doesn't have to any relation between the different comonads T_i, but often there will be. Commonly the set is a partially ordered set (as it will be in our example), and this often leads us to some relationship between the comonads related to indexes which are ordered by the poset ordering. In our example we have an inclusion from the comonads associated smaller elements into comonads associated with bigger elements.

When we have a comonad in Kleisli form, we can define a coKleisli category $Kl(T)$ over the original category $\mathcal{C}$. The objects are the same as in $\mathcal{C}$, while the morphisms from A to B are the morphisms from TA to B in the original category. The identities are the maps ϵ_A, and composition of $f : TA \to B$ and $g : TB \to C$ is given by $g \bullet f = g \circ f^*$.

2.2. Pebble Games and Finite Model Theory

Model Theory is a central part of the foundations of computer science and logic. It is the study of standard mathematical structures from a logical view point. The central object is that of a model for a theory. A theory is a set of sentences (or propositions) in some formal language, with a model (or structure) for the theory being an interpretation of the language in some universe where all the sentences of the theory hold. In finite model theory, unsurprisingly, we restrict our study to those models which have a finite universe, called finite structures. Note that while technically a structure $\mathfrak{A}$ consists of the universe and an interpretation function for the signature, I will often refer to the universe of $\mathfrak{A}$ as $\mathfrak{A}$, for example choosing an element $a \in \mathfrak{A}$ when technically I should say $a \in U_{\mathfrak{A}}$ for $U_{\mathfrak{A}}$ the universe of $\mathfrak{A}$. Throughout this report I'll assume all signatures are purely relational, so don't have constants or function symbols.

An important tool for understanding finite models is Ehrenfeucht-Fraïssé (EF) games. This is a game of two players, called the Spoiler and the Duplicator, played on two finite structures. The game is played in rounds, for a normally finite number of rounds. In the most general version, each round is played as follows, for structures $\mathfrak{A}$ and $\mathfrak{B}$:

1. The Spoiler picks a structure, either $\mathfrak{A}$ or $\mathfrak{B}$.
2. The Spoiler then chooses an element of the universe of the structure they chose, say $a \in \mathfrak{A}$.
3. The Duplicator then picks an element of the other structure, say $b \in \mathfrak{B}$.

We define a_i and b_i to be the elements of $\mathfrak{A}$ and $\mathfrak{B}$ picked during round i. We then define a partial relation $f : \mathfrak{A} \to \mathfrak{B}$ sending a_i to b_i for each i. This doesn't even have to eb a function, as the Spoiler could pick the same element twice and the Duplicator could pick different elements in response. Then the Duplicator's aim is choose an element at each round so this partial relation f is always a partial bijection, and an isomorphism - that is, an isomorphism on the elements which it is defined upon. In model theory an isomorphism is one which respects the structure of the relations in the signature, i.e for every relation R, where we say $R^{\mathfrak{A}}$ is R interpreted in $\mathfrak{A}$ and likewise for $\mathfrak{B}$, we need:

$$\forall x_1 \dots x_n \; R^{\mathfrak{A}}(x_1, \dots, \; x_n) \iff R^{\mathfrak{B}}(f(x_1), \dots, \; f(x_n))$$

Note here that all the x_i are in the domain of f, and so have been picked at some point in the game. We say the Duplicator has a winning strategy for the k-round EF game if they can always play such that this condition holds, and if not then the Spoiler has a winning strategy. This captures the notion of the Duplicator trying to trick the Spoiler into believing the two structures are equal, even when they may not be.

We consider a restriction of this game called the existential EF game. In this restriction the Spoiler and Duplicator respectively are only allowed to play on one structure each throughout the entire game. So for example the Spoiler will play on $\mathfrak{A}$ in every round and likewise with the Duplicator playing only on $\mathfrak{B}$.

Now we define a fragment of first order logic called FO[k], which is the fragment of all formulae of quantifier rank at most k. The quantifier rank, qr of a formula is the maximum depth of nested quantifiers within the formula. We can define this by structural induction:

- $qr(\phi) = 0$ for all atomic ϕ
- $qr(\phi \wedge \psi) = qr(\phi \vee \psi) = max(qr(\phi), qr(\psi))$
- $qr(\neg\phi) = qr(\phi)$
- $qr(\exists x\phi) = qr(\forall x\phi) = qr(\phi) + 1$

We can then restrict this fragment to be only positive existential sentences, so we disallow negation and the universal quantification. We define this fragment as it shows us the usefulness of the existential EF game, by the following result (derived from a result in [6]):

Theorem 2.2.1 (Ehrenfeucht-Fraïssé) *Let $\mathfrak{A}$ and $\mathfrak{B}$ be two structures in a relational vocabulary. Then the following are equivalent:*

- $\mathfrak{A} \vDash \mathfrak{B}$ *on FO[k]: for all ϕ in FO[k], $\mathfrak{A} \vDash \phi \implies \mathfrak{B} \vDash \phi$*
- *The Duplicator has a winning strategy in the existential k-round game with the Spoiler playing on $\mathfrak{A}$ and the Duplicator on $\mathfrak{B}$*

I'll now introduce a modified version of this game called the pebble game, which is similar to the EF game, but with some key differences. Firstly, instead of just choosing

elements of the structures, the Spoiler and Duplicator pick a pebbles from a finite set $\{p_1, p_2, \ldots p_k\}$ (effectively an index or label), and then choose an element of the structure to place it on. They can pick a pebble already chosen in a previous move, which will move the pebble from the previously chosen element of the structure to the newly chosen element. However, the Duplicator always has to pick the same pebble each round as the Spoiler chose that round. This means that even if the game has many more than k rounds, there can only be k chosen elements of each structure. The other difference is that we ask for strategies on an unlimited number of rounds, so here the limited access is due to the number of pebbles, rather than the number of rounds in the EF game.

The idea of a winning strategy for the Duplicator is similar to the EF game, although here we define the partial relation by mapping elements with the same pebble placed on them to each other. So if in round i the Spoiler places pebble k on an element a, and the the Duplicator places the same pebble on an element b, then f will relate a to b. The Duplicator has a winning strategy if after every round this partial relation is always a partial isomorphism, as before. In this game, as we have unlimited rounds, this partial relation will often change its value on the same elements as the pebbles are moved around, but it will only ever be defined on k elements where k is the number of pebbles we're playing with. This game also has an existential version where the Spoiler and Duplicator are assigned structures at the beginning of the game and will only play on their assigned structures.

The motivation for the set up of this game is due to the following proposition, which again can be found in [6]:

Theorem 2.2.2 *The following are equivalent:*

- *Every sentence of the existential positive k-variable fragment of first-order logic satisfied by $\mathfrak{A}$ is also satisfied by $\mathfrak{B}$.*
- *Duplicator has a winning strategy in the existential k-pebble game with the Spoiler playing on $\mathfrak{A}$ and the Duplicator on $\mathfrak{B}$.*

When we're viewing the games in the existential form, a strategy for the Duplicator is effectively a function which, given a sequence of moves by the Spoiler on the structure $\mathfrak{A}$, returns the correct element to pick in the next round. In the k-round EF game this sequence is equivalent to a list of elements of $\mathfrak{A}$ of length at most k, and in the pebble game it is equivalent to a list of elements of $\mathfrak{A}$ each paired with a pebble index $i \in \{1, \ldots, k\}$. This is the form employed in [1] to describe the universes of the structures resulting from applying the comonad functors, and what conditions there are on a winning strategy existing.

Now we understand these games, and the ideas behind comonads, we can combine the two to describe the main theoretical focuses of this project.

2.3. The EF Comonad

We use as our base category $\mathcal{R}(\sigma)$. The objects are structures over some purely relational signature σ (hence with no functions or constants), and morphisms are functions of σ-

structures which preserve all relations. That means for any relation $R \in \sigma$ - with $f : \mathfrak{A} \to \mathfrak{B}$ - then $\forall a_1 \dots a_n \in \mathfrak{A}\ R^{\mathfrak{A}}(a_1, \dots, a_n) \iff R^{\mathfrak{B}}(f(a_1), \dots, f(a_n))$. The following definitions come from [2]. We'll construct the indexed comonad first in Kleisli form, although it will be useful to construct both forms. We'll need a map on the objects of $\mathcal{R}(\sigma)$, counit maps for every object and the coextension operations. For the k round EF game, with the spoiler playing on a structure $\mathfrak{A}$, we define the map $\mathbb{E}_k\mathfrak{A}$ by specifying the universe of $\mathbb{E}_B\mathfrak{A}$, and how we interpret the relations of the signature in this new structure. The universe is all possible (non-empty) sequences of plays on $\mathfrak{A}$ of length up to k, and we can define the mappings ϵ_A as mapping a sequence to its last element. Then for a relation R, we say:

$$R^{\mathbb{E}_k\mathfrak{A}}(p_1, \dots,\ p_m) \iff (\forall i, j :\ p_i \sqsubseteq p_j \vee p_j \sqsubseteq p_i) \wedge R^{\mathfrak{A}}(\epsilon_A(p_1), \dots,\ \epsilon_A(p_m))$$

where by $\sqsubseteq$ we mean the prefix relation on sequences. The first condition then implies that the plays form a sequence under the prefix ordering. The reason we define the relation in this way is so that a strategy must respect this relation to be a winning strategy. To see this, consider $a, b \in \mathfrak{A}$ to be related by some binary relation R interpreted in $\mathfrak{A}$, and suppose they are both picked at some point in a sequences of plays s by the Spoiler, with b picked before a. If s_b is the sequence up to b and likewise for a, then $s_b \sqsubseteq s_a$, and $R(\varepsilon_A(s_b), \varepsilon_A(s_a))$, so we satisfy the conditions of the pebble relation. Then for a strategy $f : \mathbb{E}_k\mathfrak{A} \to \mathfrak{B}$, to be a homomorphism in $\mathcal{R}(\sigma)$ means it must respect this relation, so $R^{\mathfrak{B}}(f(s_b), f(s_a))$. As f is a strategy, $f(s_b)$ is just the element the Duplicator will pick at that point in the sequence (in response to the Spoiler picking b), and likewise with $f(s_a)$ and a. Hence the elements picked at these points will be related in $\mathfrak{B}$ as f is a homomorphism, so on a and b the partial map defined during this game will be a homomorphism. As all this was for an arbitrary relation and elements, we can see the idea behind this definition enabling us to prove the theorem below. This prefix condition effectively weakens the relation, and hence the condition on having a winning strategy, to be only concerned about relations between elements in the same sequence of plays.

Finally, we define the coextension, for $f : \mathbb{E}_k\mathfrak{A} \to \mathfrak{B}$, as $f^*([a_1, \dots,\ a_j]) = [f(s_1), \dots,\ f(s_j)]$ where $s_i = [a_1, \dots,\ a_i]$, the first i elements of the sequence. We can understand this as if f is a strategy for the Duplicator telling them what to play after each sequence of moves by the Spoiler, then f^* will give us the entire sequence of moves the Duplicator will have made up until that point, including the move they should make in the current round. The proof that this forms a comonad can be found in [2].

The standard triple form can then be constructed from this, or given directly. To extend $\mathbb{E}_k$ to a functor we define $\mathbb{E}_k f([a_1, \dots,\ a_j]) = [f(a_1), \dots,\ f(a_j)]$, and the comultiplication is initial segments function: $\delta_A([a_1, \dots,\ a_j]) = [s_1, \dots,\ s_j]$ where again $s_i = [a_1, \dots,\ a_i]$. Note that we can as before as construct the coKleisli category $Kl(\mathbb{E}_k)$ using this comonad, and a morphism $f : \mathfrak{A} \to \mathfrak{B}$ in $Kl(\mathbb{E}_k)$ is a morphism $f : \mathbb{E}_k\mathfrak{A} \to \mathfrak{B}$ in $\mathcal{R}(\sigma)$. We can now state the following result, from [2]:

Theorem 2.3.1 *The following are equivalent:*

- $f : \mathbb{E}_k\mathfrak{A} \to B$ *defines a winning strategy for the Duplicator in the existential k-*

round EF game.

- *f is a morphism of $\mathcal{R}(\sigma)$, i.e. a homomorphism of σ-structures.*

This tells us that this comonad formulation entirely captures the k-round existential EF game, which is exactly what we wanted. Now we turn to the slightly more complicated example of the pebble comonad

2.4. The Pebble Comonad

In the pebble game, our sequences of plays can be arbitrarily long, so we no longer have the restriction on the length of the sequence. However, we need the sequences of plays to include the information about which pebble is picked at each round. Hence, the indexed pebble comonad $\mathbb{P}_k$ represents the set of possible plays by the Spoiler on a structure $\mathfrak{A}$, when playing with k pebbles. As before we'll need to construct another σ-structure $\mathbb{P}_k\mathfrak{A}$, and describe how relations are interpreted in this new structure. Most of the following definitions come from [1].

We define the universe of $\mathbb{P}_k\mathfrak{A}$ to be the set $([k] \times \mathfrak{A})^+$, i.e. the set of non-empty sequences of plays on $\mathfrak{A}$, where a play is represented by the pebble n chosen and the element of $\mathfrak{A}$ that pebble is placed on (where $[k]$ is the set $\{1, 2 \dots k\}$). We'll define a new function:

$\varepsilon_A : \mathbb{P}_k\mathfrak{A} \to \mathfrak{A}$

$\varepsilon_A([(p_1, a_1), (p_2, a_2) \dots (p_m, a_m)]) = a_m$

For a relation R in the signature, we interpret $R^{\mathbb{P}_k\mathfrak{A}}(s_1 \dots,\ s_n)$ for $s_i \in \mathbb{P}_k\mathfrak{A}$ as being true just in the case that the following three conditions are satisfied:

- $\forall i, j\ s_i \sqsubseteq s_j \vee s_j \sqsubseteq s_i$
- $R^{\mathfrak{A}}(\epsilon_A(s_1), \dots,\ \epsilon_A(s_n))$
- $\forall i, j$ (assuming $s_i \sqsubseteq s_j$), then the last pebble played in s_i must not appear in the suffix of s_i in s_j

All the definitions so far are almost identical to those for the EF comonad, but now we have this extra condition on the pebbles placed in the plays we're relating. This condition is due to the finite nature of the number of pebbles we have. Only the most recent move using a pebble p_i is relevant to the current position in the game, so we can effectively ignore wherever the pebble has been placed previously in the game if it has subsequently been moved. So a strategy only needs to worry about the current state of all the pebbles - not where they've been previously - in deciding the next move.

The coextension operation for a function $f : \mathbb{P}_k\mathfrak{A} \to \mathfrak{B}$ is defined similarly to in the EF game: $f^*([(p_1, a_1), \dots,\ (p_n, a_n)]) = [(p_1, f(s_1)) \dots,\ (p_n, f(s_n))]$ where $s_i = [(p_1, a_1) \dots,\ (p_i, a_i)]$. As before this will give us the sequence of moves the Duplicator would play following the strategy f in response to a sequence of plays by the Spoiler, but here these plays have to have the pebble index chosen as well.

Again we give the triple form directly: To extend $\mathbb{P}_k$ to a functor from $\mathcal{R}(\sigma)$ to itself, we need to define how it acts on functions, so for $f : A \to B$ we define:

$\mathbb{P}_k\mathfrak{A}(f) : \mathbb{P}_k\mathfrak{A} \to \mathbb{P}_k\mathfrak{B}$

$\mathbb{P}_k\mathfrak{A}(f)([(p_1, a_1), (p_2, a_2) \dots (p_m, a_m)]) = [(p_1, f(a_1)), (p_2, f(a_2)) \dots (p_m, f(a_m))]$

It remains to define the comultiplication δ_A:

$\delta_A : \mathbb{P}_k\mathfrak{A} \to \mathbb{P}_k\mathbb{P}_k\mathfrak{A}$

$\delta_A(s) = [(p_1, s_1), (p_2, s_2) \dots (p_m, s_m)]$ where the s_is are as before.

We won't prove it here, but in [1] there is a proof that this is a functor, and that ε_A is the counit, δ_A the comultiplication, and hence the triple $(\mathbb{P}_k, \varepsilon_A, \delta_A)$ forms an indexed comonad on $\mathcal{R}(\sigma)$.

As before we get a result concerning the conditions for a function to be a winning strategy: [1]:

Theorem 2.4.1 *The following are equivalent:*

- $f : \mathbb{P}_k\mathfrak{A} \to B$ *is a winning strategy for the Duplicator in the existential k-pebble game.*
- f *is a morphism of* $\mathcal{R}(\sigma)$*, i.e. a homomorphism or* σ*-structures.*

This tells us the comonadic formulation entirely captures the pebble game within a category theoretic concept, as also in the case of the EF game. Hence when in Chapter 4 I provide functions of the same form as above and proofs that these functions are homomorphisms, this will be equivalent to providing a winning strategy for the Duplicator, (and hence we won't be able to provide such a proof if there is no winning strategy).

2.5. Idris, Dependent Types and Proofs

Idris, as mentioned above, is a general purpose functional programming language developed by Edwin Brady at the University of St Andrews [4]. Dependent types are built on the work of Per Martin-Löf and his intuitionistic type theory [7], and have been included in several languages such as Coq [3] and Agda [8]. Idris is a language resulting from essentially asking the question *"What if Haskell had full dependent types?"*. It is compiled, eagerly evaluated, and its "headline" improvement over a language like Haskell or ML is that it includes dependent types. It also contains a proof system which can be used to ask for and provide proofs for properties of your values, hence allowing for even more precision in specifying your programs behaviour in just the type signature by including proofs of properties as arguments of the function. All this combines to give a very expressive system with a lot of capability for specificity in writing your code, so that if it type checks, it probably works. Here I'll explain the two key features mentioned, and briefly explain some other parts which I will use for the implementation of the pebble comonad. Throughout this section most of the code examples come either from the Prelude definitions of Idris, or are adapted from samples in [5].

2.5.1. Dependent Types

Firstly, dependent types. In the simplest terms, this is a type parametrised by some other element. The easiest example is that of Lists of a given length, which in the

literature are called vectors. If we define for `n` a natural number the type `Vect n` to be lists of length `n`, then we can more explicitly type some basic list operations as follow (note, type signatures use a single colon, as opposed to the double colon in Haskell):

```
head : Vect n A -> A
tail : Vect (n+1) A -> Vect n A
map  : (A -> B) -> Vect n A -> Vect n B
(++) : Vect n A -> Vect m B -> Vect (n+m) B
```

This would allow us to check if we weren't defining these function correctly, as the Idris type checker would notice that the list returned by the function may have the wrong length. To be more precise about how this works, in Idris types are first order, the same as functions and values. Hence they can be passed to functions, and returned from functions. To refer to our example, `Vect` is really a function typed as follows:

```
Vect : Nat -> Type -> Type
```

where `Nat` is the type of natural numbers (including 0). Note that in the typing of `(++)` the return type is parametrised by the types of the inputs, and this is a key part of how dependent types work in Idris. With functions like these, we are able to use the implicit argument carried in the type of the inputs as part of the function body.

To avoid an infinite stack of `Type`s, and to resolve the question "what type does `Type` have?", Idris has a hierarchy of universes. So `Nat : Type1 : Type2 : ...`. We can't refer to these levels of types in our code explicitly, but when type checking and compiling the code, Idris will work out which level any declaration of `Type` is. I won't talk more about this facet of Idris, but I note it here to clear any doubts about how it might work.

2.5.2. Proofs

The second key feature I will be using is Idris' in-built proof system. This system uses the standard features of typed functional programming in a novel way to encapsulate proofs within the language. Roughly speaking, instantiated type signatures act as proofs of propositions, and data constructors act as properties of their arguments. For a basic example, let's use non-empty lists, and the head operation. We can define a property `NonEmpty xs` for a list `xs` as follows:

```
data NonEmpty : List a -> Type where
    IsNonEmpty (x::xs) -> NonEmpty (x::xs)
```

This can be translated as the property `NonEmpty` can hold of any `List` of the form `(x::xs)`, i.e. any non-empty list. That is, to give a proof of `NonEmpty xs`, we need to use the constructor `IsNonEmpty` on the non-empty list xs. We could then type the function `head`:

```
head : (xs : List a) -> (NonEmpty xs) -> a
```

This would mean in applying `head`, we need to provide a proof that the value we apply it to will not be an empty list. In fact, it would be frustrating to have to give a proof for every instance, so Idris defines it slightly differently:

```
head : (xs : List a) -> {auto ok : NonEmpty xs} -> a
head []      {ok=IsNonEmpty} impossible
head (x::xs) {ok=p}  = x
```

A few things to explain here: First, the `auto` is to tell Idris to try and automatically construct a proof that `xs` is non empty without you having to provide one. On the second line, the `impossible` signifies that we can't apply this definition of `head` to a non-empty list, as a proof that `[]` is non-empty is impossible to supply. We have this line so Idris still knows the function is total. Then the third line is our familiar definition.

The reason we would want to define our function like this, rather than removing the second line, is that we want all our functions to be total. This means the Idris compiler will always expand our definitions fully, and be able to type check more complicated expression in the knowledge that in doing so, it won't get in an infinite loop with a partial function. We can mark a function as total by writing `total` before the function name in the type signature. Idris will then try check that the function is total, using conservative constraints. Due to the halting problem it won't always be able to decide whether the function is total, but if all calls inside the definition are to total functions, and either the function is non-recursive or is recursive but the recursive argument can be shown to have strictly decreased at each call, then Idris decides the function is total.

Above we defined our own property a list may have. There's only one built in property of the language, that is equality of values. We can imagine it defined as:

```
data (=) : a -> b -> type where
Refl : x = x
```

As is this is a property like `NonEmpty`, we can treat it as a type in our definitions, or more specifically proofs. As an example, consider the statement that mapping `g` over a list then mapping `f` over the result is equal to mapping `f` composed with `g` over the same list. We would write this in Idris as:

```
mapFusion : (f : b -> c) -> (g : a -> b) -> (l : List a) ->
    map f (map g l) = map (f . g) l
```

This is just a statement of what we want to prove. To actually prove it, we need to provide a definition of a total function with that type signature. To rewrite proofs in Idris (i.e to substitue facts into out equalities) we an operation `rewrite prf in expr`. If `expr` is some property on `x`, and `prf` is an equality proof that shows `x = y`, then `rewrite prf in expr` will be a proof that `expr` holds for `y`.

Finally, we need to note that Idris allows us to construct inductive proofs. For example here, we prove this statement by induction on the length of the list, which Idris recognises as a proof as it is a total function which type-checks:

```
mapFusion f g []    = Refl
mapFusion f g (x::xs) =
    let inductiveHypothesis = mapFusion f g xs in
        rewrite inductiveHypothesis in Refl
```

The first line is clear, as Idris can easily expand the definition of `map` on the empty list to get `[]` on both sides, then we just need to use `Refl`. The third line uses the inductive hypothesis to get a proof of `mapFusion f g xs`, i.e a proof saying `map f (map g xs) = map (f . g) xs`. Then the Refl on the 4th line is stating `map f (map g (x::xs)) = map f (map g (x::xs))`. Idris will expand out the definition of `map` (as the definition is inductive), so we get

```
(f (g x)) :: (map f (map g xs)) = (f (g x)) :: (map f (map g xs))
```

The `rewrite` changes the `(map f (map g xs))` into `map (f . g) xs`. Then Idris will apply the definition of `map` in reverse, knowing that `f (g x) = (f . g) x`, to get the statement we wanted.

This proof is the standard way we might prove such a statement. Idris enables us to prove most things about common functions (a lot of standard results are already proved, and in fact all the examples above came from the Idris Prelude.List module). If we can't prove that a statement is true in Idris, but know (or believe) it to be true, we can use the function `believe_me`, which provides a proof for any statement. This should only be used when you know the statement is true, otherwise we may believe our program is correct when it isn't.

Idris also has internalised a version of proof by contradiction, using the type `Void`. This type has no elements, and idris provides a function `void :: Void -> a` which allows us to give a value of any type if we have a value of type void. This is a slightly more advanced version of marking cases as impossible which we did earlier in the head function, where Idris isn't able to work out a case is impossible immediately, so we to construct an element of type void by using the the way we can prove a contradiction. This means either constructing a proof that two things which aren't equal are equal (for example a proof that `True = False`), or constructing a value of a type with an implementation of the Uninhabited interface. This interface provides a function `uninhabited` which takes a value of the type and returns Void, and is a way of marking certain types (and hence certain properties of values) as impossible. For example for the `Nat` type we have `Uninhabited (Z = S n) uninhabited Refl impossible`. We can construct these instances ourselves, and this makes it easier to show certain general properties are impossible to obtain.

2.5.3. Extra Features

Idris has a few other small features I will be utilising, which I will briefly go over here:

Sigma Types Also known as dependent pairs, or dependent tuples. A sigma type is a tuple like construction, but where the type of each element can depend on the value of the previous elements. For example, we might want the type of all `Vect`s from our definition earlier. Then we'd use:

```
AllVectors : Type
AllVectors =
    DPair Type $ \t =>
    DPair Nat $ \n =>
    Vect n t

-- this will type check
aVector : AllVectors
aVector = (Int ** 3 ** [0,1,2])

-- this won't type check
notAVector : AllVectors
notAVector = (Int ** 3 ** [a,b,c,d])
```

The definition of `AllVectors` means that each member of the type is a triple with the first element a type, the second a `Nat`, and the third a `Vect` with the length and type of the first 2 elements (and hence it depends on their values). In `notAVector`, we can see that the vector is of the wrong length and has the wrong type of elements, so this wouldn't type check. We can access the elements of the tuple using pattern matching, just like standard tuples.

Dependent Records Records in Idris are similar to Haskell records, but we're only allowed one constructor which must be supplied all parameters, so the accessors functions are total. A basic record might be as follows:

```
record Class where
    constructor MkClass
    Size : Nat
    Members : Vect Size String

myClass : Class
```

```
myClass = MkClass 2 ["Robert", "Samson"]
```

Note that even in standard records, the parameters can depend on each other as in a dependent type signature. A dependent record is one in which several parameters can be given in the type of the record. Suppose we wanted classes of different sizes to have different types. Then we'd give:

```
record SizedClass (size : Nat) where
    constructor MkSizedClass
    Members : Vect size String

myClass : SizedClass 2
myClass = MkSizedClass ["Ambramsky", "Dawar"]

-- this won't typecheck
notClass : SizedClass 2
notClass = MkSizedClass ["Ambramsky", "Dawar", "Wang"]
```

Codata Types and Explicit Laziness As mentioned earlier, Idris has strict, eager evaluation. However, it still allows the use of laziness or infinite datatypes through the concept of a codata type. This is a datatype definition, but where certain arguments to the constructors of the datatype can be marked by `Inf`, to demonstrated they might be infinite, and hence Idris won't evaluate them until necessary. This means we can get lazy evaluation, if we mark explicitly those part which should be lazily evaluated.

The standard example is that of infinite lists, known in Idris as streams. They have the following definition:

```
data Stream : Type -> Type where
    (::) : elem -> Inf (Stream elem) -> Stream elem
```

Notice the `Inf` marking the `Stream elem` argument to the `(::)` constructor, meaning that Idris won't evaluate that argument until it needs to. We could then define some basic infinite lists and operations on them as follows:

```
ones :: Stream Nat
ones = 1 :: ones

map :: (a -> b) -> Stream a -> Stream b

zipWith : (f : a -> b -> c) -> (xs : Stream a) -> (ys : Stream b) -> Stream c
zipWith f (x::xs) (y::ys) = f x y :: zipWith f xs ys
```

Note as `(::)` is the only constructor for the `Stream` type, we can pattern match exclusively on that pattern.

The "with" Rule Sometimes we want to pattern match on the result of in intermediate computation when writing a function. The with rule allows us to do this, by computing the intermediate computation and allowing you to write your definition by matching on the result's pattern. For example consider a filter function on the Vector type we defined above:

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
filter p [] = ( _ ** [] )
filter p (x :: xs) with (filter p xs)
    | ( _ ** xs' ) = if (p x) then ( _ ** x :: xs' ) else ( _ ** xs' )
```

The result is a dependent type, as we won't know what the length of the filtered vector will be. Here the `with` computes `filter p xs`, and we can the pattern match on the result, which will be of the form `( _ ** xs' )`. The rest of the function is then the expression after the `=`.

3. The Implementation of the Game Comonads

The relevant code files are in Appendix A, but all necessary material will explained as it is introduced here. I'll introduce each module in turn, and then in chapter 4 I give examples.

3.1. Category Theory

First, I'll explain my implementation of the relevant category theory concepts in Idris. One of the ways of viewing a category is as having a function which takes two objects A and B of the category, and returns the set of morphisms between those two objects, $Hom(A, B)$ (known as the hom-set). Hence, to completely define a category, we need the type of objects, the hom-set function, identity functions for each hom-set, and a composition function which composes morphisms. We'll also need these functions to satisfy the standard axioms concerning composition being associative, and the identity morphisms being the identity of composition. I've chosen this method of representing a category as it's the most common used in other functional programming techniques, and the syntax lines up nicely with the notation of category theory. It also automatically captures part of category theory which other implementations might lack, such as every morphism having a defined domain and codomain. The definition for functors, and then comonads, follows in a similar way.

Figure 3.1.: Category Theory Definitions

```
record RCategory (obj : Type) where
    constructor RCategoryInfo
    mor  : obj -> obj -> Type
    idd  : {p: obj} -> mor p p
    comp : {a : obj} -> {b : obj} -> {c : obj} -> mor b c -> mor a b -> mor a c

record RFunctor (obj: Type) (cat : RCategory obj) where
    constructor RFunctorInfo
    func : obj -> obj
    fmap : {a : obj} -> {b : obj} -> (mor cat) a b -> (mor cat) (func a) (func b)

record RComonad (obj : Type) (cat : RCategory obj) where
    constructor RComonadInfo
    comon  : RFunctor obj cat
    counit : {a : obj} -> (mor cat) a ((func comon) a)
    comult : {a : obj} -> (mor cat) ((func comon) a) ((func comon) ((func comon) a))
```

Often we might use interfaces rather than records to implement this kind of abstract mathematical idea, as is done in Haskell, and then implementations of the interfaces are instances of that type of structure. I use records here as there's less constraints on what instances can be.

In Idris, only data or type constructors can be instances of an interface, but the implementation of our instances of these concepts often won't be data-types or type constructors, but just types created directly using Idris's type system, hence we must use records. Interfaces have the advantage of being able to provide natural extensions, so any monad must be an instance of a functor. We can't do this explicitly with records, but we can use an equivalent concept by making the record for the extended type require an parameter of the base type. For example, a comonad requires a functor as an argument to be constructed.

A quick note that the arguments given in curly braces as seen in **??** are those implicit to the function. Normally we wouldn't need to write out these implicit arguments, but it helps Idris type check the code correctly, and when we implement instances of these records we won't need to fit to the definitions with the implicit arguments, just the explicit ones. Also, the accessor methods for the record are named after the argument which needs to be accessed, so `mor cat` returns the morphism function from the category `cat`.

The key definition is that of the indexed comonad, which I provide in standard and Kleisli form. It's key to note here is I'm using a slightly different definition, where there's a condition on the indexing set, which needs to hold for the index to produce a valid comonad. This is because in Idris natural numbers start at 0, but we want them to start at 1 when indexing pebbles, as we can't have 0 pebbles, or 0 rounds in our game. We could effectively view this indexing set as the set of pairs of a natural number and a proof it is non zero, which is equivalent to the set non-zero natural numbers, and then by currying the pair of arguments can be seen as two separate arguments, which is easier for us to reason about.

Figure 3.2.: Indexed Conditional Comonad

```
record RIxCondComonad (obj : Type) (cat : RCategory obj)
    (ind : Type) (cond : ind -> Type) where
    constructor RIxCondComonadInfo
    indcomon : (k : ind) -> {pf : cond k} -> RFunctor obj cat
    counit  : {a : obj} -> {k : ind} -> {p : cond k} -> (mor cat) ((func (indcomon k
        {pf = p})) a) a
    comult  : {a : obj} -> {k : ind} -> {p : cond k} -> (mor cat) ((func (indcomon k
        {pf = p})) a) ((func (indcomon k {pf = p})) ((func (indcomon k {pf = p})) a))

record RIxCondComonadKleisli (obj : Type) (cat : RCategory obj) (ind : Type) (cond :
     ind -> Type) where
    constructor RIxCondComonadKleisliInfo
    mapping : (k : ind) -> {pf : cond k} -> obj -> obj
    counit  : {a : obj} -> (k : ind) -> {p : cond k} -> (mor cat) (mapping k {pf = p}
        a) a
    coextend : {a, b : obj} -> (k : ind) -> {p : cond k} -> (mor cat) (mapping k {pf =
        p} a) b -> (mor cat) (mapping k {pf = p} a) (mapping k {pf = p} b)
```

3.2. Model Theory, Graphs and Structures

I take two approaches to implementing the necessary model theory concepts in Idris, one a more complicated version of the other. First we consider the simplification of $\mathcal{R}(\sigma)$ where the signature σ has just 1 binary relation. Then this category is equivalent to the category of graphs, with a model for the signature being a graph where the vertices of e graph are the universe, and edges form the interpretation of the relation. This simplification means we don't need to deal

with signatures and interpretations, and instead deal just with the edge relation directly. The universes of the models for this relation will be non-empty sets, but they could be infinite (hence we won't be dealing strictly with finite models here). We need infinite models as the universe of plays for the pebble comonad is infinite, as there's no bound on the length of a sequence of plays. To be able to have infinite universes, we'll need the notion of a possibly infinite list, for which we'll use the implicit laziness described in section 2.5. Theses sets of edges will need to be non empty also, so we create a new list-like data type which can't be constructed without any elements, and so must be non-empty.

Figure 3.3.: The Graph type

```
data NEStream : (elem : Type) -> Type where
    Sing : (x : elem) -> NEStream elem
    (:>:) : (x : elem) -> (xs : Inf (NEStream elem)) -> NEStream elem

Rel : Type -> Type
Rel t = (t,t) -> Bool

Graph : Type
Graph =
    DPair Type $ \v =>
    DPair (NEStream v) $ \vs =>
    DPair (Rel v) $ \es =>
    Eq v
```

We use a dependent tuple for our Graph type, which first takes the type of the vertices in the graph. It then takes the non-empty possibly infinite list of vertices, the relation defining the edges, and an instance of the equality type class for the type we're using. This means our type has a built in equality which we require explicitly, which is necessary for comparing elements in later parts of the implementation.

Now we want to be able to build the type of morphisms between two graphs. The best way of encoding this is asking for a function of the correct type, and then a proof that the function obeys the property required to be a morphism. This is the property that $\forall a, b \in G$, if (a, b) is an edge then (fa, fb) is also an edge in the target graph. Thinking about this proof in terms of the type system of Idris, it's effectively a function which takes two elements $a, b : G$, and a proof that they form an edge, and returns a proof that (fa, fb) is an edge. The edges will be defined by a relation as we said earlier, so we encode the idea of a proof that (a, b) is an edge for a relation $es : (t, t) \to Bool$ by asking for a proof that $True = es(a, b)$. All standard boolean logical operations can be translated into this type level boolean logic. While type theory works using intuitionistic logic rather than classical logic, we can still derive standard logical properties in the type level easily (for example we can prove that $True = a \;\&\&\; b \to True = a$). We do this simply by case exhaustion of all possible a and b.

As we define the composition and identity morphisms, we need to provide proofs for the resulting morphisms that they have this property. I'll explain the example of the composition function, as it serves a good example of when the Idris proof system works well. To prove the statement required, we need to use the proofs that the individual morphisms are correct. Remembering these proofs are functions which take arguments and return proofs, we see that for any two elements `x, y : t1` such that `True = e1 (x, y)`, we use the first proof to show that `True = e2 (vmap1 x, vmap1 y)`, then the second proof to show that `True = e3 (vmap2 (vmap1 x), vmap2 (vmap1 y)`, which is exactly what we needed to prove.

Figure 3.4.: The Category of Graphs

```
data IsGraphMorph : (f: t1 -> t2) -> (e1 : Rel t1) -> (e2 : Rel t2) -> Type where
    IsGraphMorphByElem :
       ((a : t1) -> (b : t1) -> True = e1 (a, b) -> True = e2 (f a, f b))
          -> IsGraphMorph f e1 e2

data Gmorph : (g1 : Graph) -> (g2 : Graph) -> Type where
    Gmor : (f: t1 -> t2) -> IsGraphMorph f e1 e2 -> Gmorph (t1 ** v1 ** e1 ** eq1) (t2
        ** v2 ** e2 ** eq2)

GidProof : (es : Rel t) -> (a, b : t) -> True = es (a, b) -> True = es (Basics.id a,
    Basics.id b)
GidProof es a b eqprf1 = eqprf1

Gid : Gmorph a a
Gid {a = (t ** vs ** es ** eq)} = Gmor Basics.id (IsGraphMorphByElem (GidProof es))

compProof : (f1 : ta -> tb) -> (f2 : tb -> tc) -> (eas : Rel ta) -> (ebs : Rel tb) ->
    (ecs : Rel tc) -> IsGraphMorph f1 eas ebs -> IsGraphMorph f2 ebs ecs -> (x, y: ta)
    -> True = eas (x, y) -> True = ecs ((f2 . f1) x, (f2 . f1) y)
compProof vmap1 vmap2 eas ebc ecs (IsGraphMorphByElem aToBEqPrf) (IsGraphMorphByElem
    bToCEqPrf) x y eqprf1 =
    bToCEqPrf (vmap1 x) (vmap1 y) (aToBEqPrf x y eqprf1)

(..) : Gmorph b c -> Gmorph a b -> Gmorph a c
(..) {a = (ta ** vas ** eas ** eqa)} {b = (tb ** vbs ** ebs ** eqb)} {c = (tc ** vcs **
    ecs ** eqc)} (Gmor vmap2 (IsGraphMorphByElem bToCEqPrf)) (Gmor vmap1
    (IsGraphMorphByElem aToBEqPrf)) =
    Gmor (vmap2 . vmap1) (IsGraphMorphByElem (compProof vmap1 vmap2 eas ebs ecs
        (IsGraphMorphByElem aToBEqPrf) (IsGraphMorphByElem bToCEqPrf)))

GraphCat : RCategory Graph
GraphCat = RCategoryInfo Gmorph Gid (..)
```

So we have our category of graphs. It's clear that the category axioms hold of these definitions:

```
idd .. (Gmor f prf) = Gmor (id . f) (compProof idd f ...)
                    = Gmor f (compProof idd f ...)

(Gmor f prf) .. idd = Gmor (f . id) (compProof idd f ...)
                    = Gmor f (compProof idd f ...)

((Gmor f prff) .. (Gmor g prfg)) .. (Gmor h prfh) = (Gmor (f . g) (compProof f g ...))
    .. (Gmor h prfh)
       = Gmor (f . g . h) ()compProof f.g ... compProof(h ...))
       = ...
       = (Gmor f prff) .. ((Gmor g prfg) .. (Gmor h prfh))
```

Note that we don't care whether the proofs are equal here, just that the functions are equal and both proofs can be constructed, hence we have the axioms as required.

The Graph category is the one we'll use when dealing with the pebble comonad, as implementing a comonad over it is simpler, and allows us to have infinite universes. For the EF comonad

however, we don't need infinite universes, so we'll consider a more complicated category closer to $\mathcal{R}(\sigma)$. In this new category, all the relations are still binary, but we allow a finite number of them. Hence this category is one parametrised by a signature which tells us how many relations there are. As all the signature needs to encode is the number of relations, we can encode in a natural number, using a type alias: `Signature = Nat`. I call this new category we're defining the category of sStructures. The objects will be structures over a certain signature, and I define them similarly to how the Graph type is defined. However, instead of a single relation, we need to provide an interpretation, which will be a function from the signature to the relations. The functions domain being a natural number can be easily modelled by the `Fin` data type, which is defined in Idris as follows:

```
data Fin : (n : Nat) -> Type where
    FZ : Fin (S k)
    FS : Fin k -> Fin (S k)
```

This is the finite set of numbers strictly less than n, or equivalently n viewed as the set $\{0, \ldots n-1\}$. The other difference will be that the set of elements in the universe which before was a non-empty stream possibly infinite) will now be a non-empty list guaranteed finite).

The definition of a structure morphism is similar to that of a graph morphism, but where we're universally quantifying over all elements of the signature, and asking that the function is a graph morphism with respect to each every relation when interpreted by the structure. This allows us to reuse the proofs that the composition and identities are morphism in the graph category by applying them to every possible relation given by our interpretation.

Figure 3.5.: The Category of Structures

```
Interpretation : Signature -> Type -> Type
Interpretation sig t = (Fin sig) -> (Rel t)

data IsStructureMorph : (sig : Signature) -> (f : t1 -> t2) -> (int1 : Interpretation
    sig t1) -> (int2 : Interpretation sig t2) -> Type where
    EmptySigStructMorph : sig = 0 -> IsStructureMorph sig f int1 int2
    ItIsStructMorph : ((k : Fin sig) -> IsGraphMorph f (int1 k) (int2 k)) ->
        IsStructureMorph sig f int1 int2

data StructMorph : {sig : Signature} -> (a1, a2 : Structure sig) -> Type where
    Smor : (f : t1 -> t2) -> IsStructureMorph sig f int1 int2 -> StructMorph (t1 ** v1
        ** int1 ** eq1) (t2 ** v2 ** int2 ** eq2)
(...)
Sid : StructMorph a a
Sid {a = (t ** vs ** rels ** eq)} = Smor Basics.id (SidProof rels)

infixr 9 ./.

(./.) : {sign : Signature} -> StructMorph {sig = sign} b c -> StructMorph {sig = sign}
    a b -> StructMorph {sig = sign} a c
(...)
StructCat : (sigma : Signature) -> RCategory (Structure sigma)
StructCat sigma = RCategoryInfo StructMorph Sid (./.)
```

3.3. The Pebble Comonad

Now we have all the relevant groundwork laid, we can define and implement the pebble comonad. As was explained in section 2.4, interpreted in the language of graphs as the structures, the functor will need to construct a new graph from the original one, with the new graph's vertices being all possible sequences of plays on the old graph's vertices, and the edges being the extended relation as previously defined. I'll then define the counit, comultiplication and the definition of the functor on morphisms, and alternatively the counit and coextension map to give the Kleisli form.

I use the `Fin` data type introduced earlier to restrict the pebble indices to the correct bound, and the type of non-empty lists. This is defined identically to the non-empty stream type demonstrated in section 2.5.3, but without the infinite label on the second argument, which means the lists must be finite. While the sequences of plays can be arbitrarily large, we don't allow infinite sequences, so this doesn't need to be a stream. To generate the new set of plays on an existing graph, the `plays` function iterates infinitely starting from the initial universe, generating plays of length $n + 1$ from plays of length n, and then concatenating these lists of plays into the outer list.

To construct the relation, the definition is as described in section 2.4, but for only binary relations. I use the `with` rule here to condition on the lengths of the two sequences, as that will change how we calculate the truth of the relation. We also need the proof of which way the lengths compare so we can correctly get the suffix of the shorter list in the longer one, which

Figure 3.6.: Generating Sequences of Plays

```
playsType : Nat -> Type -> Type
playsType pebs t = NEList (Fin pebs, t)

[playsTypeEq] Interfaces.Eq t => Interfaces.Eq (playsType n t) where
    (==) (Singl x) (Singl y) = x == y
    (==) (x:<:xs) (y:<:ys) =
    if x==y then xs==ys else False
    (==) _ _ = False

iterate : (a -> a) -> a -> NEStream a
(...)
--A unit is required here as all the lists could be empty
concatNESofList : (unit : t) -> NEStream (List t) -> NEStream t
(...)
concatNESofNES : NEStream (NEStream t) -> NEStream t

plays : Eq t => (pebs:Nat) -> {auto ok : IsSucc pebs} -> (xs: NEStream t) -> (NEStream
    (playsType pebs t))
plays (S pebs) xs = concatNESofNES (iterate (uplength xs pebs) (initial pebs xs))
    where uplength : NEStream t -> (pebs:Nat) -> NEStream (playsType (S pebs) t) ->
        NEStream (playsType (S pebs) t)
         uplength zs pebs ys = concatNESofNES (map (\els => concatNESofList (Singl (FZ,
             (head xs))) (map (\el => map (\p => (restrict pebs (toIntegerNat p),
             el):<:els)[0..pebs]) zs)) ys)
         initial : (pebs:Nat) -> NEStream t-> NEStream (playsType (S pebs) t)
         initial pebs ys = concatNESofList (Singl (FZ, (head xs))) (map (\y => map (\p
             => (Singl (restrict pebs (toIntegerNat p), y))) [0..pebs]) ys)
```

requires us to know that the shorter list is actually shorter. Having the function stated in this way means when we come to prove this function obeys certain properties with respect to its inputs xs and ys we can use the `with` rule with the exact same statement in the proof function. Doing this allows Idris to automatically work out what we need to prove in each case in our proof function (automatically evaluating the original function with the information about the result of the intermediate computation performed in the `with` statement).

The implementation of how the pebble functor acts on morphisms is straightforward, as described in section 2.4, where it simply maps the function of the second element in each pair of the sequence of plays given as input. Having constructed the pebble functor over the category of graphs, it remains for me to describe the counit and comultiplication. The counit is simple to implement, as it's the second element in the last pair of the sequence of plays. The proof that this is a graph morphism is again simple, as for the sequences of plays to be relations by the pebbled relation implies among other things that the second of the last pairs of the two sequences are related, exactly as required. The comultiplication's implementation is more complicated, where for a sequence of plays xs we zip the list of pebbles played in xs with the list of initial segments of the xs, as described in section 2.4. Proving this is a graph morphism would be quite difficult and fiddly, but the proof would work the same as described in [1]. The coextension operation is simply a combination of the functor's action on morphisms and the comultiplication, where for a function f applied to xs we get the initial segments of xs, map f over it and then zip it together with the original list of pebbles played in xs. Proving this is a graph morphisms would be effectively the same as proving both the result of the pebble functor acting on morphisms, and the comultiplication, are graph morphisms. Hence we have defined the full indexed pebble comonad within Idris.

Figure 3.7.: The Pebble Relation

```
pebblesRelSuffix : Eq t => (as : pebPlaysType pebs t) -> (bs : pebPlaysType pebs t) ->
    Bool
pebblesRelSuffix xs ys with (isLTE (length xs) (length ys))
    | Yes prf =
       (NonEmptyList.isPrefixOf xs ys) && (isNothing (NonEmptyList.find ((==)
           (Basics.fst (last xs))) (map fst (drop ((-) (length ys) (length xs)) ys))))
    | No contra =
       (isPrefixOf ys xs) && (isNothing (NonEmptyList.find ((==) (Basics.fst (last
           ys))) (map fst (drop ((-) {smaller = notLTEImpliesLT _ _ contra} (length xs)
           (length ys)) xs))))


pebblesRel : Eq t => Rel t -> Rel (pebPlaysType pebs t)
pebblesRel r (xs, ys) = (pebblesRelSuffix xs ys) && (r (snd (last xs), snd (last ys)))
```

3.4. The EF Comonad

For the EF comonad, as the structure is simpler, and we only need finite universes, I will implement it over the structure category defined in section 3.2, so we possibly have more than one relation. My implementation follows the same pattern as the pebble comonad implementation, with a few key differences.

The first difference is the type of the universe of the new structure resulting from the object mapping. We need something equivalent to non-empty lists of elements in the original universe, with length at most n where n is the number of rounds. There would be a number of different ways of implementing this. We could employ a pattern similar to the `Fin` data type, but where the arguments are elements or lists, and we start at singleton lists. This would be a combination of the non-empty list type and the `Fin` type. However, the problem with creating a new data type is a lot of standard proofs which we might expect to be available for the already defined lists aren't available, so we'll need to prove everything from scratch. Also, the `Fin` type is very hard to deal with, as it's often unclear what type an element n is, as it can be `Fin k` for any $k > n$, and we'd get the same problem with this data type. We'll still need to use the `Fin` type, but I have created a data type with only uses it to bound the lists length, and then the sequence

Figure 3.8.: The EF Comonad Universe and Plays

```
data IsFSucc : Fin k -> Type where
    ItIsFSucc : IsFSucc (FS k)

data EFplaysType : (n: Nat) -> (t : Type) -> Type where
    MkPlays : (k : Fin (S n)) -> {auto ok : IsFSucc k} -> Vect (finToNat k) t ->
        EFplaysType n t

[EFplaysTypeEq] Eq t => Eq (EFplaysType n t) where
    (==) (MkPlays k xs) (MkPlays j ys) = k == j && toList xs == toList ys

efplays : (bound : Nat) -> \{auto ok : IsSucc bound\} -> NEList t -> NEList
     (EFplaysType bound t)
efplays (S Z) \{ok = p\}  xs = initial xs
efplays (S (S k)) \{ok = p\} xs = myIt 1 (LTESucc (LTESucc LTEZero)) (uplength xs)
     (initial xs)
    where myIt : (start : Nat) -> LTE (S start) (S (S k)) -> ((n : Nat) -> NEList
        (EFplaysType n t) -> NEList (EFplaysType (S n) t)) ->
    NEList (EFplaysType start t) -> NEList (EFplaysType (S (S k)) t)
    myIt start LTEZero f xs impossible
    myIt start (LTESucc startLTProof) f xs with (isLTE (S start) (S k)) proof p
    | Yes prfSStart =
        ap (rewrite sym (plusMinusProof start (S (S k)) (lteSuccRight startLTProof)) in
        (map (efWeakenN ((-) {smaller = lteSuccLeft (LTESucc startLTProof)} (S (S k))
            start)) xs))
        (assert_total (myIt (S start) (LTESucc prfSStart) f (f start xs)))
    | No contra =
        ap (map efWeaken (rewrite sym (lteToEqual start (S k) (startLTProof) (\p =>
            contra p)) in xs))
        (rewrite sym (lteToEqual (S start) (S (S k)) (LTESucc startLTProof) (\(LTESucc
            p) => contra p)) in (f start xs))
```

of plays is held in a vector, a built in type which we described in section 2.5 which will be easier to work with. The data type also carries a proof that the length is a successor, so the list is non empty.

We also need a new implementation of the play generating function. Now this function will be total and strict, rather than the lazy version in the pebble comonad's implementation, as we have a finite universe instead of an infinite one. We no longer need a infinite iterate function for non-empty streams, and instead we define a new bounded iterating function using a `with` clause as our guard of the loop. This with clause also provides us the proof of the value of the guard (which is limiting the length of plays generated), so we can use it to prove we're generating plays of the correct length (and hence the correct type). When using this function, we need to keep the types as strict as possible so the iterator always knows that the lengths of the plays being passed to it are within the bound. We then weaken the types of all the plays up to the correct bound using the `efWeakenN`, which can be seen as the injection described in [2] from $\mathbb{E}_k\mathfrak{A} \hookrightarrow \mathbb{E}_j\mathfrak{A}$ for $k < j$. The lifting of the relation to the EF universe is of course also different from the pebble universe, as we don't have the condition on the pebble indices in the two lists. The minus function on natural numbers needs a proof that its first argument is greater than or equal to the second argument, so it's output can still be a natural number.

The object mapping and functor's action on morphisms are easily defined, as is the counit. For the coextension we define an initial function for our plays, which effectively acts like the standard initial segments function on vectors while keeping track of the lengths of the lists in the `MkPlays` constructor.

Figure 3.9.: Extracts from the EF game coextension definitions

```
playInits : (k : Nat) -> (n : Nat) -> {auto ok1 : IsSucc k} -> {auto ok2 : IsSucc n} ->
    {auto lt : LTE n k} -> Vect n t -> Vect n (EFplaysType k t)
playInits Z n {ok1 = ItIsSucc} xs impossible
playInits (S j) Z {ok1 = p} {ok2 = ItIsSucc} xs impossible
playInits (S j) (S Z) {ok1 = p} {ok2 = q} [x] = [MkPlays (FS FZ) [x]]
playInits (S Z) (S (S k)) {ok1 = p} {ok2 = q} {lt = LTESucc r} (x ::xs) = absurd r
playInits (S (S j)) (S (S k)) {ok1 = p} {ok2 = q} {lt = LTESucc r} (x::xs) =
(MkPlays (FS FZ) [x]) :: (map (\(MkPlays l ys) => MkPlays (FS l) (x::ys)) (playInits (S
    j) (S k) {lt = r} xs))

morphExtend {j = bnd} morph (MkPlays (FS k) {ok = p} xs) = MkPlays (FS k) {ok = p} (map
    morph (playInits (S bnd) (S (finToNat k)) {lt = finToNatLTBound (S bnd) k} xs))

coextensionEF {sigma = S sig} {s1 = (t1 ** vs1 ** interp1 ** eqt1)} {s2 = (t2 ** vs2 **
    interp2 ** eqt2)} (S j) {ok = p} (Smor morph (ItIsStructMorph strutPrf)) = Smor
    (morphExtend morph) (ItIsStructMorph prf)
   where  prf : (k : Fin (S sig)) -> IsGraphMorph (morphExtend morph) (efRel (interp1
       k)) (efRel (interp2 k))
          prf k = IsGraphMorphByElem (coextPrf {j = j} (interp1 k) (interp2 k) morph
              (strutPrf k))
```

4. Examples

I'll give some examples of the background structure from finite model theory and category theory, and give an example of a provably winning strategy in the EF comonad, accompanied by a proof that the structures are equivalent.

4.1. Background Examples

To give a worked example using the category theory module, we can consider the classic case of the category of Types, the one called Hask in Haskell (although this might strictly speaking not be a proper category) and other languages with monads and functors. As a lot of these concept are also built into Idris, it easy to define such a category:

```
arr : Type -> Type -> Type
arr t1 t2 = t1 -> t2

TypeCategory : RCategory Type
TypeCategory = RCategoryInfo arr id (.)
```

We can also give a demonstration of a comonad (and hence a functor also), inspired by this stack overflow question [11]. This is the idea that the non-empty list functor forms a comonad on the category of types, with the counit being the `head` function and the comultiplication being the single element list constructor. The comonad then takes the following form:

```
nonEmptyListFunctor : RFunctor Type TypeCategory
nonEmptyListFunctor = RFunctorInfo NEList map

nonEmptyListComonad : RComonad Type TypeCategory
nonEmptyListComonad = RComonadInfo nonEmptyListFunctor head Singl
```

We consider another example, drawing on the idea of a partially ordered set as a category, where the morphisms between any two elements or present only when the domain element is less than or equal to the codomain element. Using natural numbers, we get the following:

```
data LessThanEqual : Nat -> Nat -> Type where
    IsLessThanEqual : (n : Nat) -> (m : Nat) -> LTE n m -> LessThanEqual n m

POSetCategory : RCategory Nat
POSetCategory = RCategoryInfo LessThanEqual lteId lteComp
```

The composition is just transitivity, and the identity just reflexivity. Then any order preserving function is a functor on this category, as follows:

```
mult2Map : LessThanEqual a b -> LessThanEqual (mult2 a) (mult2 b)
mult2Map (IsLessThanEqual a b prfab) = IsLessThanEqual (mult2 a) (mult2 b) (prf a b
    prfab)
    where prf : (x : Nat) -> (y : Nat) -> LTE x y -> LTE (mult2 x) (mult2 y)
    prf Z y prfxy = LTEZero
    prf x Z prfxy = rewrite ltZimpliesZ prfxy in LTEZero
```

```
    prf (S n) (S m) prfxy = let rec = prf n m (fromLteSucc prfxy) in LTESucc (LTESucc
        rec)

mult2Functor : RFunctor Nat POSetCategory
mult2Functor = RFunctorInfo mult2 mult2Map
```

We can also give a preliminary of an example of a graph:

```
lt : Rel Nat
lt (a,b) = a < b

[testEqNat] Eq Nat where
(==) a b = (toIntegerNat a) - (toIntegerNat b) == 0

NatGraph : Graph
NatGraph = (Nat ** listToNEStream [1,2,3,4,5] ** lt ** testEqNat)
```

Here we've created a pretty basic directed graph which has an edge from every number to those bigger than it. `listToNEStream` does the obvious thing in converting a list to a non-empty stream, but the function requires its input to be a non-empty list, by having an implicit argument that is a proof that the list is non-empty (as in the definition of the `head` function for lists). The definition of the type class needs to be in terms of some other kind of equality, so we use integer equality. While this is obviously the same as the standard equality for `Nat` build into Idris, we need a named equality implementation to pass it as an argument for our dependent tuple.

Now let us try to create another graph and a homomorphism between them. We can define the second graph, and then provide the morphism:

```
NatGraph : Graph
NatGraph = (Nat ** listToNEStream [1,2,3,4,5] ** lt ** testEqNat)

ltchar : Rel Char
ltchar (n,m) = toNat n < toNat m

[testEqString] Eq Char where
(==) n m = (toNat n) == (toNat m)

myChar : Type
myChar = Char

CharGraph : Graph
CharGraph = (myChar ** listToNEStream ['1','2','3','4','5'] ** ltchar ** testEqString)

testGraphMorph : Gmorph CharGraph NatGraph
testGraphMorph = Gmor (toNat) (IsGraphMorphByElem prf)
where prf : (a : Char) -> (b : Char) -> True = ltchar (a,b) ->
True = lt (toNat a, toNat b)
prf a b prf1 = prf1
```

This is a basic example, where we've effectively replicated the first graph with characters in place of numbers. Hence the proof that the function is a graph morphisms is very simple, as it just falls out of how the graphs are defined.

4.2. Demonstration of Morphism of a Winning Strategy in the EF comonad

Figure 4.1.: The Strategy Function

```
Ex1Strategy : EFplaysType 2 Ex1Universe -> Ex1Universe
Ex1Strategy (MkPlays (FS FZ) [x]) = One
Ex1Strategy (MkPlays (FS (FS FZ)) [x, y])
    with ((==) @{Ex1Equality} (Ex1Next x) y, (==) @{Ex1Equality} (Ex1Next y) x)
    | (False, False) = Three
    | (False, True) = Four
    | (True, False) = Two
    | (True, True) = One
```

One of the uses of the implementation of these game comonads is we can then use the internal proof system of Idris to check whether a winning strategy can be proved to be a morphism, and hence prove within the language the property of two structures as stated in the two theorems 2.4.1 and 2.2.1. Here I will provide a toy example of such a winning strategy with proof within Idris.

For our example, the two structures we use are a chain of length 4, and a cycle of length 4, $\mathrm{L}_4 := \{l_1, l_2, l_3, l_4\}$ and $C_4 := \{c_1, c_2, c_3, c_4\}$ respectively. In both structures we have a single binary relation R. In the chain we have $\forall i \in \{1, 2, 3\}\ l_i R l_{i+1}$ and for the cycle $\forall i \in \{1, 2, 3, 4\}\ c_i R c_{i+1\ mod\ 4}$. We play the EF game with two rounds on these structures, with the spoiler playing on L_4 and the duplicator on C_4. This means we need a strategy for the duplicator, which will be a morphism $f : \mathbb{E}_2 L_4 \to C_4$. The strategy will be as follows:

- Regardless on where the spoiler plays in the first round, we play on the first element of the cycle, c_1.
- We respond to the spoilers second move by conditioning on the relation between the first and second move:
    1. If the spoiler's second move is directly after its first move (for example l_1 then l_2) then we choose c_2, as this will preserve the relation.
    2. If the spoiler's second move is directly before its first (l_3 followed by l_2) then we choose c_4, and as C_4 is a cycle we preserve the relation.
    3. Otherwise, the two elements or not related, so we choose c_3 as it's not related to c_1.

It's trivial to see our strategy will work as we have just exhausted all possible cases. The intuition behind why the cycle satisfies all sentences of 2 or fewer variables that the chain does is that with only 2 variables we can't tell the difference between these a chain and a cycle. It can be be proved the other direction of this equivalence with a similar strategy, so in fact these two structures are equivalent in the 2 variable fragment of FO, but I won't give a proof here.

Now we need to implement these ideas into Idris. First we need to define these 2 structures, their universes and relations and notions of equality. We can use the same universe here (a set of 4 elements), and just provide a different relation for each structure. The implementation of the strategy works using a `with` statement which gives us the conditioning on the relation between the first two moves as described above. One of these results is technically impossible, but we can use that in our proof to eliminate that case. The proof then follows the case exhaustion strategy of our original proof, but with a finer detail so Idris can see all the steps. We need a

Figure 4.2.: The Strategy Proof

```
stratProof : (a : EFplaysType 2 Ex1Universe) -> (b : EFplaysType 2 Ex1Universe) ->
    (True = efRel @{Ex1Equality} (Ex1Rel1 FZ) (a, b)) -> True = Ex1Rel2 FZ (Ex1Strategy
        a, Ex1Strategy b)
stratProof (MkPlays (FS FZ) [x]) (MkPlays (FS FZ) [y]) prf =
    let contra = ex1relTrueImpliesNotEqual x y (conjunctsTrueL prf) in
    let diction = conjunctsTrueL (conjunctsTrueR {b = ((==) @{Ex1Equality} x y &&
        True)} prf) in void (contra diction)
stratProof (MkPlays (FS (FS FZ)) (x::[xl])) (MkPlays (FS FZ) [y]) prf with ((==)
    @{Ex1Equality} (Ex1Next x) xl, (==) @{Ex1Equality} (Ex1Next xl) x) proof p
    | (False, False) =
       let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (xl, y)} prf in
       let x1' = ex1rel1ImpliesNext xl y x1 in
       let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
           (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
       let x3 = pairsSplitR p in
       let test = equalityCongruence1 x1' x2 x3 in sym (equalityReflexivity1 y test)
    | (False, True) = Refl
    | (True, False) =
       let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (xl, y)} prf in
       let x1' = ex1rel1ImpliesNext xl y x1 in
       let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
           (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
       let x3 = pairsSplitR p in
       let test = equalityCongruence1 x1' x2 x3 in sym (equalityReflexivity1 y test)
    | (True, True) = void (ex1NoMutualNext x xl (pairsSplitL p) (pairsSplitR p))
stratProof (MkPlays (FS FZ) [y]) (MkPlays (FS (FS FZ)) (x::[xl])) prf = ...
    --symetrical to above case
stratProof (MkPlays (FS (FS FZ)) (x::[xl])) (MkPlays (FS (FS FZ)) (y::[yl])) prf =
    let x1 = ex1rel1ImpliesNext xl yl (conjunctsTrueL prf {a = Ex1Rel1 FZ (xl, yl)}) in
    let x2 = ex1EqCorrect xl yl (conjunctsTrueL (conjunctsTrueR (conjunctsTrueR prf)))
        in
    void (ex1NextChanges xl (sym (trans x1 (sym x2))))
```

proof function that takes two sequences of plays of length at most two (elements of $\mathbb{E}_2 L_4$), and also takes a proof that these sequences are related in the chain's extended relation, and returns a proof that the strategy's answers to these 2 sequences are related also. We have the following cases:

- If the two plays are both of length 1, then it is actually impossible for them to be related by the extended relation. This is because if they are to a prefixes of one another then they have to be equal, but then in the chain no element is related to itself. So for this case in our proof function we can create a contradiction and use the void function to give a total result to this branch.
- If one play is a singleton and the other is length 2, then we can condition whether the 2 elements in the longer play are related in any way. As we know the element in the single play is the same as the first element in the double play by the 2 plays being related, this covers all possible cases. In some results we construct a contradiction as it is impossible for the 2 lists to be related. For example, if we had $[l_1, l_3]$ and $[l_1]$, then $[l_1] \sqsubseteq [l_1, l_3]$, but l_1 is not related to l_3 so these lists aren't related. To be specific, in these cases Idris asks

for a proof that `True = False`, and by the fact that the arguments lead to contradictions we can construct a proof of this fact directly rather than using the void function.

- For the case where the arguments are all possible, due to Idris' case analysis it can work out what we're trying to prove is already true, so we only need to provide a `Refl` statement.
- If both plays are length two again it is impossible that they are related, as they must be equal but then their last elements can't be related.

There are several intermediate proofs which I use in this proof, such as the fact that no element is equal to its next element, or that no element is related to itself. These can all be easily defined by case exhaustion, but this can be tiresome as we end up with 16 cases for any property of 2 elements of the universe. I chose to implement the proof in this way - using a data type where each element of the universe was a different constructor - as it allows Idris to do such case exhaustion as this. Without being able to condition on the different cases the objects can come in, you need to prove stuff which holds about the general case, which can often be hard as Idris won't expand the general case. That is, if the data type wasn't defined inductively or with many different constructors, any property needs to be almost immediately obvious from the construction for it to be easily provable in Idris. Even if the data type is defined inductively (for example we could have used `Fin 5` which would have been a 4 element finite set), if your properties aren't provable from this induction - as most of the properties I used in the intermediate proofs here - you will still have difficulty. I'll discuss in more depth the pros and cons of Idris and its proof system in the next chapter.

5. Conclusions

In this project I've described and implement two game comonads in Idris. I've shown the variety of ways in which Idris can be used to describe concepts in model theory and category theory, as well as the potential for it to be used as a proof assistant to verify results about structures in fragments of first order logic.

5.1. The Power of Idris

Using Idris to express and prove the properties required has its benefits, as well as its downsides. Idris makes proving some properties, such as those which can be proved inductively, or of types with multiple constructors, simple and understandable. I found during the course of this project however that when it comes to proving properties aren't provable in this way, it often becomes very difficult to construct a proof. I also had problems actually writing out many of the proofs even when it was clear what I wanted to do, due to the lack of documentation on how the proof system works in detail, and rules which seemed to apply differently in certain contexts. Some of this is inevitably down to my inexperience with Idris, but often if I came across a problem, my only course of action was to ask for help online, as the documentation and error messages could often be unclear or confusing.

While the proof system of Idris can be hard to work with, the flexibility and expressibility of dependent types were vital in being able to construct this project. Once I was sure I had written the correct type, Idris was able to catch many subtle errors I would have missed by pointing out type errors. If I was to do a similar project or work to this in the future, I would look forward to the power of the dependent type system. The more properties of the concept we're expressing I could contain in the dependent type signatures, as opposed to asking for proofs, the easier the program was to type. For example, if there was a way of defining the type of graph morphisms in a structural way which didn't require a proof to be provided, then a lot of work would be saved in having to write out proofs of simple facts which we know to be true, but can often be tiring and fiddly to express in Idris.

To summarise, the ability to express in dependent types what we needed to was vital, but the proof system of Idris I found to be often difficult to work with, and is very conservative in trying to work things out by itself. I don't have experience with other proof languages such as Coq or Agda, but their proof systems might have been better suited to this task. However, I chose Idris due to its familiar Haskell-like syntax, which may make it more readable to more people than if this project had been written in Agda.

5.2. Personal Development

Throughout this project, many of the skills and knowledge learned in my Computer Science courses have been applied and built upon. The exploration of the category theory wouldn't have been possible without the Categories, Proofs and Processes course given by Samson Abramsky, and courses in Functional Programming and Logic and Proof in the earlier years were a good foundation on which to learn about Idris and finite model theory, neither of which I'd used

or seen before this project. I've learned how to use the Idris language effectively (often after multiple rewrites of the same pieces of code), and the benefit of designing in the abstract the software's approach and structure before programming the specifics. I regret not being able to take a course on Type Theory as my work in Idris has sparked that interest for me, as well as interest in exploring finite model theory and the links between the structure approach of category theory, and the power and expressibility approach of model theory and logic.

5.3. Further Work

There's lots of potential for further work in this area. A few possibilities:

- Expressing more concepts from category theory and finite model theory, specifically the unfolding comonad based on modal logic also found in [2], either in Idris or another programming language.
- Developing proofs of more complicated strategies and properties of the logical structures we've discussed.
- Improving Idris's proof system's usability and readability, or building inside Idris a more solid framework for proving properties about logical structures, with help in writing out proofs.
- Converting proof based properties to structurally typed properties as discussed above would make many parts of the project simpler, if appropriate structural data types could be found.

A. Code Listings

The entire directory can be found at https://github.com/RobertKirk/Graph-Comonads-from-Pebble-Games/tree/master/src, but I present the relevant implementation files here as reference.

A.1. EFComonad.idr

```
module EFComonad
import Data.Fin
import Data.Vect
import src.ProofHelpers
import src.NonEmptyStream
import src.NonEmptyList

import src.RCategories
import src.Graphs

%access public export
%default total

data EFplaysType : (n: Nat) -> (t : Type) -> Type where
MkPlays : (k : Fin (S n)) -> {auto ok : IsFSucc k} -> Vect (finToNat k) t ->
    EFplaysType n t

Uninhabited (EFplaysType Z t) where
uninhabited (MkPlays FZ {ok = ItIsFSucc} xs) impossible
uninhabited (MkPlays (FS k) {ok = p} xs) impossible

efWeaken : EFplaysType n t -> EFplaysType (S n) t
efWeaken (MkPlays FZ {ok = IsFSucc} xs) impossible
efWeaken (MkPlays (FS j) {ok = p} xs) = MkPlays (FS (weaken j)) (rewrite sym
    (weakenPreservesToNat (FS j)) in xs)

efWeakenN : (m : Nat) -> EFplaysType n t -> EFplaysType (n + m) t
efWeakenN m (MkPlays FZ {ok = IsFSucc} xs) impossible
efWeakenN m (MkPlays (FS j) {ok = p} xs) = MkPlays (FS (weakenN m j)) (rewrite sym
    (weakenNPreservesToNat m (FS j)) in xs)

[EFplaysTypeEq] Eq t => Eq (EFplaysType n t) where
(==) (MkPlays k xs) (MkPlays j ys) = k == j && toList xs == toList ys

initial : NEList t -> NEList (EFplaysType 1 t)
initial xs = map (\x => MkPlays (FS FZ) [x]) xs

uplength : NEList t -> (n : Nat) -> NEList (EFplaysType n t) -> NEList (EFplaysType (S
    n) t)
```

```
uplength xs n plays = concatNELofNEL (map (\(MkPlays len play) => (map (\el => MkPlays
    (FS len) (el::play)) xs)) plays)

efplays : (bound : Nat) -> {auto ok : IsSucc bound} -> NEList t -> NEList (EFplaysType
    bound t)
efplays Z {ok = ItIsSucc} xs impossible
efplays (S Z) {ok = p}  xs = initial xs
efplays (S (S k)) {ok = p} xs = myIt 1 (LTESucc (LTESucc LTEZero)) (uplength xs)
    (initial xs)
    where myIt : (start : Nat) -> LTE (S start) (S (S k)) -> ((n : Nat) -> NEList
        (EFplaysType n t) -> NEList (EFplaysType (S n) t)) -> NEList (EFplaysType start
        t) -> NEList (EFplaysType (S (S k)) t)
    myIt start LTEZero f xs impossible
    myIt start (LTESucc startLTProof) f xs with (isLTE (S start) (S k)) proof p
        | Yes prfSStart = ap (rewrite sym (plusMinusProof start (S (S k)) (lteSuccRight
            startLTProof)) in
            (map (efWeakenN ((-) {smaller = lteSuccLeft ((LTESucc startLTProof))} (S (S
                k)) start)) xs)) (assert_total (myIt (S start) (LTESucc prfSStart) f (f
                start xs)))
        | No contra = ap (map efWeaken (rewrite sym (lteToEqual start (S k))
            (startLTProof) (\p => contra p)) in xs))
            (rewrite sym (lteToEqual (S start) (S (S k)) (LTESucc startLTProof)
                (\(LTESucc p) => contra p)) in (f start xs))

efRelPrefix : Eq t => Vect j t -> Vect k t -> Bool
efRelPrefix {j = n} {k = m} xs ys with (isLTE n m)
    | Yes prf = isPrefixOf xs ys
    | No contra = isPrefixOf ys xs

efRel : Eq t => Rel t -> Rel (EFplaysType bound t)
efRel r ((MkPlays FZ {ok = ItIsFSucc {k = FZ}} xs), (MkPlays FZ {ok = ItIsFSucc} ys))
    impossible
efRel r (p1, (MkPlays FZ {ok = ItIsFSucc} ys)) impossible
efRel r ((MkPlays (FS j) {ok = pj} xs), (MkPlays (FS k) {ok = pk} ys)) = r ((vlast xs),
    (vlast ys)) && efRelPrefix xs ys

efInterp : Eq t => (bound : Nat) -> (int : Interpretation sig t) -> Interpretation sig
    (EFplaysType bound t)
efInterp bound int rel = efRel (int rel)

EFComonadObj : (bound : Nat) -> {auto ok : IsSucc bound} -> Structure sig -> Structure
    sig
EFComonadObj Z {ok = ItIsSucc} strut impossible
EFComonadObj (S k) {ok = p} (t ** vs ** interp ** eqt) =
    (EFplaysType (S k) t **
    efplays (S k) vs **
    efInterp (S k) interp **
    EFplaysTypeEq)

efMap : (bound : Nat) -> {auto ok : IsSucc bound} -> (t1 -> t2) -> EFplaysType bound t1
    -> EFplaysType bound t2
efMap Z {ok = ItIsSucc} morph xs impossible
efMap (S k) {ok = p}  morph (MkPlays l xs) = MkPlays l (map morph xs)
```

```
EFComonadMorph : (bound : Nat) -> {auto ok : IsSucc bound} -> StructMorph s1 s2 ->
    StructMorph (EFComonadObj bound s1) (EFComonadObj bound s2)
EFComonadMorph Z {ok = ItIsSucc} smorph impossible
EFComonadMorph (S k) {ok = p} (Smor f prf) = Smor (efMap (S k) {ok = p} f) ?efMorphProof

EFFunctor : (bound : Nat) -> {auto ok : IsSucc bound} -> RFunctor (Structure sigma)
    (StructCat sigma)
EFFunctor Z {ok = ItIsSucc} impossible
EFFunctor (S k) {ok = p} = RFunctorInfo (EFComonadObj (S k)) (EFComonadMorph (S k))

counitFunc : EFplaysType (S k) t -> t
counitFunc (MkPlays FZ {ok = ItIsFSucc} xs) impossible
counitFunc (MkPlays (FS j) {ok = p} xs) = vlast xs

counitEF : {sigma : Signature} -> {s : Structure sigma} -> (bound : Nat) -> {auto ok :
    IsSucc bound} -> StructMorph (EFComonadObj bound s) s
counitEF {sigma = sig} Z {ok = ItIsSuc} impossible
counitEF {sigma = Z} {s = (t ** vs ** interp ** eqt)} (S k) {ok = p} = Smor
    EFComonad.counitFunc (EmptySigStructMorph Refl)
counitEF {sigma = (S sig)} {s = (t ** vs ** interp ** eqt)} (S k) {ok = p} = Smor
    EFComonad.counitFunc (ItIsStructMorph prf)
   where  intprf : (a, b : EFplaysType (S k) t) -> True = efRel r (a, b) -> True = r
       (counitFunc a, counitFunc b)
          intprf (MkPlays FZ {ok = ItIsFSucc} xs) p proof1 impossible
          intprf q (MkPlays FZ {ok = ItIsFSucc} xs) proof1 impossible
          intprf (MkPlays (FS lx) {ok = p} xs) (MkPlays (FS ly) {ok = q} ys) proof1 =
              conjunctsTrueL proof1
          prf : (rel : Fin (S sig)) -> IsGraphMorph EFComonad.counitFunc (efInterp (S
              k) interp rel) (interp rel)
          prf rel = IsGraphMorphByElem intprf

playInits : (k : Nat) -> (n : Nat) -> {auto ok1 : IsSucc k} -> {auto ok2 : IsSucc n} ->
    {auto lt : LTE n k} -> Vect n t -> Vect n (EFplaysType k t)
playInits Z n {ok1 = ItIsSucc} xs impossible
playInits (S j) Z {ok1 = p} {ok2 = ItIsSucc} xs impossible
playInits (S j) (S Z) {ok1 = p} {ok2 = q} [x] = [MkPlays (FS FZ) [x]]
playInits (S Z) (S (S k)) {ok1 = p} {ok2 = q} {lt = LTESucc r} (x ::xs) = absurd r
playInits (S (S j)) (S (S k)) {ok1 = p} {ok2 = q} {lt = LTESucc r} (x::xs) =
(MkPlays (FS FZ) [x]) :: (map (\(MkPlays l ys) => MkPlays (FS l) (x::ys)) (playInits (S
    j) (S k) {lt = r} xs))

lastOfPlaysInitsIsList : (m : Nat) -> (n : Nat) -> {auto ok1 : IsSucc m} -> {auto ok2 :
    IsSucc n} -> {auto lt : LTE n m} -> (xs : Vect n t) ->
vlast {n = n} {ok = ok2} (playInits m n xs) = MkPlays (natToFin n (S n)) {ok =
    isSuccToIsFSucc n ok2} (vectInj (finToNatToFin2 n) xs)
lastOfPlaysInitsIsList Z n {ok1 = ItIsSucc} xs impossible
lastOfPlaysInitsIsList (S n) Z {ok2 = ItIsSucc} xs impossible
lastOfPlaysInitsIsList (S j) (S Z) {ok1 = ItIsSucc} {ok2 = ItIsSucc} {lt = LTESucc p}
    [y] = ?lastPlaysProofCase1
lastOfPlaysInitsIsList (S Z) (S (S k)) {lt = LTESucc p} (x::xs) = absurd p
lastOfPlaysInitsIsList (S (S j)) (S (S k)) {lt = LTESucc p} (x::xs) =
    ?lastPlaysProofCase2
```

```
morphExtend : ((EFplaysType (S j) t1) -> t2) -> EFplaysType (S j) t1 -> EFplaysType (S
    j) t2
morphExtend {j = bnd} morph (MkPlays FZ {ok = ItIsFSucc} xs) impossible
morphExtend {j = bnd} morph (MkPlays (FS k) {ok = p} xs) = MkPlays (FS k) {ok = p} (map
    morph (playInits (S bnd) (S (finToNat k)) {lt = finToNatLTBound (S bnd) k} xs))

coextPrf : Eq t1 => Eq t2 => (rel1 : Rel t1) -> (rel2 : Rel t2) -> (morph : EFplaysType
    (S j) t1 -> t2) ->
IsGraphMorph morph (efRel rel1) rel2 -> (a : EFplaysType (S j) t1) -> (b : EFplaysType
    (S j) t1) ->
True = efRel rel1 (a, b) -> True = efRel rel2 (morphExtend morph a, morphExtend morph b)
coextPrf rel1 rel2 morph gmorphPrf (MkPlays FZ {ok = ItIsFSucc {k = FZ}} xs) pys relPrf
    impossible
coextPrf rel1 rel2 morph gmorphPrf xys (MkPlays FZ {ok = ItIsFSucc} ys) relPrf
    impossible
coextPrf rel1 rel2 morph (IsGraphMorphByElem gmorphPrf) (MkPlays (FS l1) xs) (MkPlays
    (FS l2) ys) relPrf with (isLTE (finToNat l1) (finToNat l2)) proof ltprf
  | Yes yesprf = ?coextLeftProof
  | No contra = ?coextRightProof

coextensionEF : {sigma : Signature} -> {s1, s2 : Structure sigma} -> (bound : Nat) ->
    {auto ok : IsSucc bound} ->
StructMorph (EFComonadObj bound s1) s2 -> StructMorph (EFComonadObj bound s1)
    (EFComonadObj bound s2)
coextensionEF Z {ok = ItIsSucc} morph impossible
coextensionEF {sigma = Z} {s1 = (t1 ** vs1 ** interp1 ** eqt1)} {s2 = (t2 ** vs2 **
    interp2 ** eqt2)} (S j) {ok = p} (Smor morph prof) = Smor (morphExtend morph)
    (EmptySigStructMorph Refl)
coextensionEF {sigma = S sig} {s1 = (t1 ** vs1 ** interp1 ** eqt1)} {s2 = (t2 ** vs2 **
    interp2 ** eqt2)} (S j) {ok = p} (Smor morph (EmptySigStructMorph reflp)) = absurd
    (sym reflp)
coextensionEF {sigma = S sig} {s1 = (t1 ** vs1 ** interp1 ** eqt1)} {s2 = (t2 ** vs2 **
    interp2 ** eqt2)} (S j) {ok = p} (Smor morph (ItIsStructMorph strutPrf)) =
Smor (morphExtend morph) (ItIsStructMorph prf)
  where  prf : (k : Fin (S sig)) -> IsGraphMorph (morphExtend morph) (efRel (interp1
      k)) (efRel (interp2 k))
  prf k = IsGraphMorphByElem (coextPrf {j = j} (interp1 k) (interp2 k) morph
      (strutPrf k))

EFIndexedComonadKleisli : {sigma : Signature} -> RIxCondComonadKleisli (Structure
    sigma) (StructCat sigma) Nat IsSucc
EFIndexedComonadKleisli = RIxCondComonadKleisliInfo EFComonadObj counitEF coextensionEF
```

A.2. PebbleComonad.idr

```
module PebbleComonad
import Data.Fin
import src.ProofHelpers
import src.NonEmptyStream
import src.NonEmptyList

import src.RCategories
import src.Graphs
```

```
%hide Stream.(::)
%access public export
%default total

-- a play with n pebbles is a list of pairs of the pebble being played each turn,
-- and the position in the list of the element the pebble is played on.
-- note the pebble start indexed at 0, so we can use (Fin pebs) here.
pebPlaysType : Nat -> Type -> Type
pebPlaysType pebs t = NEList (Fin pebs, t)

[pebPlaysTypeEq] Interfaces.Eq t => Interfaces.Eq (pebPlaysType n t) where
    (==) (Singl x) (Singl y) = x == y
    (==) (x:<:xs) (y:<:ys) =
    if x==y then xs==ys else False
    (==) _ _ = False

-- the Non empty (infinite) stream of plays with pebs number of pebbles on the stream xs
pebPlays : Eq t => (pebs : Nat) -> {auto ok : IsSucc pebs} -> (xs: NEStream t) ->
     (NEStream (pebPlaysType pebs t))
pebPlays Z      xs {ok = ItIsSucc} impossible
pebPlays (S pebs) xs = concatNESofNES (iterate (uplength xs pebs) (initial pebs xs))
    where  uplength : NEStream t -> (pebs:Nat) -> NEStream (pebPlaysType (S pebs) t) ->
        NEStream (pebPlaysType (S pebs) t)
           uplength zs pebs ys = concatNESofNES (map
           (\els => concatNESofList
           (Singl (FZ, (head xs)))
           (map (\el => map (\p => (restrict pebs (toIntegerNat p), el):<:els)
               [0..pebs]) zs))
           ys)
           initial : (pebs:Nat) -> NEStream t-> NEStream (pebPlaysType (S pebs) t)
           initial pebs ys = concatNESofList (Singl (FZ, (head xs))) (map (\y => map
               (\p => (Singl (restrict pebs (toIntegerNat p), y))) [0..pebs]) ys)

pebblesRelSuffix : Eq t => (as : pebPlaysType pebs t) -> (bs : pebPlaysType pebs t) ->
    Bool
pebblesRelSuffix xs ys with (isLTE (length xs) (length ys))
    | Yes prf = (NonEmptyList.isPrefixOf xs ys) && (isNothing (NonEmptyList.find ((==)
        (Basics.fst (last xs)))
    (map fst (drop ((-) (length ys) (length xs)) ys))))
    | No contra = (isPrefixOf ys xs) && (isNothing (NonEmptyList.find ((==) (Basics.fst
        (last ys)))
    (map fst (drop ((-) {smaller = notLTEImpliesLT _ _ contra} (length xs) (length ys))
        xs))))

pebblesRel : Eq t => Rel t -> Rel (pebPlaysType pebs t)
pebblesRel r (xs, ys) = (pebblesRelSuffix xs ys) && (r (snd (last xs), snd (last ys)))

PebComonadObj : (pebs:Nat) -> {auto ok : IsSucc pebs} -> Graph -> Graph
PebComonadObj Z g {ok = ItIsSucc } impossible
PebComonadObj pebs (t ** vs ** e ** eqt) {ok = p} =
    ((pebPlaysType pebs t) **
    (pebPlays {ok = p} pebs vs) **
    (pebblesRel e) **
```

```
    pebPlaysTypeEq)

pairMapRight : (t2 -> t3) -> (t1, t2) -> (t1, t3)
pairMapRight f (a, b) = (a, f b)

pebmap : (t1 -> t2) -> (pebPlaysType n t1) -> (pebPlaysType n t2)
pebmap vmap xs = map (pairMapRight vmap) xs

pebmapPreservesLength : (f : t1 -> t2) -> (xs : pebPlaysType n t1) -> length (pebmap f
    xs) = length xs
pebmapPreservesLength f xs = mapPreservesLengthNel (pairMapRight f) xs

pebMapLastandVmapCommute : (xs : pebPlaysType pebs t1) -> (vmap : t1 -> t2) -> vmap
    (snd (last xs)) = snd (last (pebmap vmap xs))
pebMapLastandVmapCommute (Singl (p,x)) vmap = Refl
pebMapLastandVmapCommute (y:<:ys)  vmap = pebMapLastandVmapCommute ys vmap

PebComonadMorphSndProof : Eq t1 => Eq t2 => (e1 : Rel t1) -> (e2 : Rel t2) -> (xs, ys :
    pebPlaysType pebs t1) ->
(vmap : t1 -> t2) -> IsGraphMorph vmap e1 e2 -> True = (e1 (snd (last xs), snd (last
    ys))) ->
True = e2 (snd (last (pebmap vmap xs)), snd (last (pebmap vmap ys)))
PebComonadMorphSndProof e1 e2 xs ys vmap (IsGraphMorphByElem vmapIsGraphMorph) e1relprf
    =
   rewrite sym (pebMapLastandVmapCommute ys vmap) in
   rewrite sym (pebMapLastandVmapCommute xs vmap) in
   vmapIsGraphMorph (snd (last xs)) (snd (last ys)) e1relprf

PebComonadMorphLTEProof : Eq t1 => Eq t2 => (pebs : Nat) -> {auto ok : IsSucc pebs} ->
    (xs : pebPlaysType pebs t1) -> (ys : pebPlaysType pebs t1) ->
(vmap : t1 -> t2) -> LTE (length xs) (length ys) ->
True = (isPrefixOf xs ys) && (isNothing (NonEmptyList.find ((==) (Basics.fst (last xs)))
(map Basics.fst (drop ((-) (length ys) (length xs)) ys)))) ->
True = pebblesRelSuffix (pebmap vmap xs) (pebmap vmap ys)
PebComonadMorphLTEProof Z {ok = ItIsSucc} xs ys vmap ltproof prf1 impossible
   PebComonadMorphLTEProof (S j) {ok = p} xs ys vmap ltproof prf1 = ?pebMorphLTEHole

PebComonadMorphGTProof : Eq t1 => Eq t2 => (pebs : Nat) -> {auto ok : IsSucc pebs} ->
    (xs, ys : pebPlaysType pebs t1) -> (vmap : t1 -> t2) ->
LTE (length ys) (length xs) ->
(True = (NonEmptyList.isPrefixOf ys xs) && (isNothing (NonEmptyList.find ((==)
    (Basics.fst (last ys)))
(map Basics.fst (drop ((-) (NonEmptyList.length xs) (NonEmptyList.length ys)) xs))))) ->
True = pebblesRelSuffix (pebmap vmap xs) (pebmap vmap ys)
PebComonadMorphGTProof Z {ok = ItIsSucc} xs ys vmap ltproof prf1 impossible
PebComonadMorphGTProof (S k) {ok = p} xs ys vmap ltproof prf1 = ?pebMorphGTHole

PebComonadMorphProof : Eq t1 => Eq t2 => (pebs : Nat) -> {auto ok : IsSucc pebs} -> (e1
    : Rel t1) -> (e2 : Rel t2) -> (vmap : t1 -> t2) -> (IsGraphMorph vmap e1 e2) -> (a
    : pebPlaysType pebs t1) -> (b : pebPlaysType pebs t1) -> True = (pebblesRel {pebs =
    pebs} e1) (a, b) -> True = (pebblesRel {pebs = pebs} e2) (pebmap vmap a, pebmap
    vmap b)
PebComonadMorphProof Z {ok = ItIsSuc} e1 e2 vmap prf a b prf2 impossible
PebComonadMorphProof (S k) {ok = p} e1 e2 vmap (IsGraphMorphByElem vmapIsGraphMorph) xs
```

```
    ys e1relprf with (isLTE (length xs) (length ys))
    | Yes yprf =
       andCombines (pebblesRelSuffix (pebmap vmap xs) (pebmap vmap ys))
       (e2 (snd (last (pebmap vmap xs)), snd (last (pebmap vmap ys))))
       (PebComonadMorphLTEProof (S k) xs ys vmap yprf (conjunctsTrueL e1relprf))
       (PebComonadMorphSndProof e1 e2 xs ys vmap (IsGraphMorphByElem vmapIsGraphMorph)
           (conjunctsTrueR e1relprf))
    | No contra =
       andCombines (pebblesRelSuffix (pebmap vmap xs) (pebmap vmap ys))
       (e2 (snd (last (pebmap vmap xs)), snd (last (pebmap vmap ys))))
       (PebComonadMorphGTProof (S k) xs ys vmap (notLTEImpliesLT _ _ contra)
           (conjunctsTrueL e1relprf))
       (PebComonadMorphSndProof e1 e2 xs ys vmap (IsGraphMorphByElem vmapIsGraphMorph)
           (conjunctsTrueR e1relprf))

PebComonadMorph : {g1, g2 : Graph} -> (pebs : Nat) -> {auto ok : IsSucc pebs} -> Gmorph
    g1 g2 -> Gmorph (PebComonadObj pebs g1) (PebComonadObj pebs g2)
PebComonadMorph Z morp {ok = ItIsSucc} impossible
PebComonadMorph {g1 = (t1 ** v1 ** e1 ** eq1)} {g2 = (t2 ** v2 ** e2 ** eq2)} pebs
    (Gmor vmap (IsGraphMorphByElem vmapIsGraphMorph)) {ok = p} =
    Gmor (pebmap vmap) (IsGraphMorphByElem prf)
    where  prf : (a : pebPlaysType pebs t1) -> (b : pebPlaysType pebs t1) -> True =
        (pebblesRel e1) (a, b) -> True = (pebblesRel e2) (pebmap vmap a, pebmap vmap b)
           prf = PebComonadMorphProof pebs {ok = p} e1 e2 vmap (IsGraphMorphByElem
               vmapIsGraphMorph)

pebbleFunctor : (pebs:Nat) -> {auto ok : IsSucc pebs} -> RFunctor Graph GraphCat
pebbleFunctor Z {ok = ItIsSucc} impossible
pebbleFunctor n {ok = p} = RFunctorInfo (PebComonadObj n {ok = p}) (PebComonadMorph n
    {ok = p})

counitPeb : {g : Graph} -> (n: Nat) -> {auto ok : IsSucc n} -> Gmorph (PebComonadObj n
    g) g
counitPeb {n = Z} {ok = ItIsSucc} impossible
counitPeb {g = (t ** vs ** es ** eq)} {n = (S k)} {ok = p} = Gmor counitFunc
    (IsGraphMorphByElem prf)
    where  counitFunc : pebPlaysType m t -> t
           counitFunc ps = snd (last ps)
           prf : (a : pebPlaysType (S k) t) -> (b : pebPlaysType (S k) t) -> True =
               (pebblesRel {pebs = (S k)} es) (a,b) -> True = es (counitFunc a,
               counitFunc b)
           prf xs ys prfaesb = conjunctsTrueR prfaesb

comultFunc : (pebPlaysType m t) -> (pebPlaysType m (pebPlaysType m t))
comultFunc plays = zip (map fst plays) (inits plays)

comultPeb : {g : Graph} -> (n: Nat) -> {auto ok : IsSucc n} -> Gmorph (PebComonadObj n
    g) (PebComonadObj n (PebComonadObj n g))
comultPeb {n = Z} {ok = ItIsSucc} impossible
comultPeb {g = (t ** vs ** es ** eq)} {n = (S k)} {ok = p} = Gmor comultFunc
    (IsGraphMorphByElem prf)
    where  prf = ?pebComultMorphProof

pebbleIndexedComonad : RIxCondComonad Graph GraphCat Nat IsSucc
```

```
pebbleIndexedComonad = RIxCondComonadInfo pebbleFunctor counitPeb comultPeb

pebmorphExtend : (pebs : Nat) -> {auto ok : IsSucc pebs} -> ((pebPlaysType pebs t1) ->
    t2) -> pebPlaysType pebs t1 -> pebPlaysType pebs t2
pebmorphExtend Z {ok = IsSucc} f plays impossible
pebmorphExtend (S k) morph plays = zip (map fst plays) (map morph (inits plays))

coextensionPeb : {g1, g2 : Graph} -> (n : Nat) -> {auto ok : IsSucc n} -> Gmorph
    (PebComonadObj n g1) g2 ->
Gmorph (PebComonadObj n g1) (PebComonadObj n g2)
coextensionPeb Z {ok = ItIsSucc} morph impossible
coextensionPeb {g1 = (t1 ** v1 ** e1 ** eq1)} {g2 = (t2 ** v2 ** e2 ** eq2)} (S k) {ok
    = p} (Gmor morph prf) = Gmor (pebmorphExtend (S k) {ok = p} morph) prf
   where prf = ?pebCoextPrf

pebbleIndexedComonadKleisli : RIxCondComonadKleisli Graph GraphCat Nat IsSucc
pebbleIndexedComonadKleisli = RIxCondComonadKleisliInfo PebComonadObj counitPeb
    coextensionPeb
```

A.3. StrategyTesting.idr

```
module StrategyTesting

import Data.Vect
import Data.Fin
import src.ProofHelpers
import src.NonEmptyList
import src.NonEmptyStream

import src.RCategories
import src.Graphs
import src.PebbleComonad
import src.EFComonad

%default total

getType : Structure sigma -> Type
getType (t ** _ ** _ ** _) = t

getList : EFplaysType n t -> List t
getList (MkPlays len xs) = toList xs

-- We assert totality so Idris know the argument in the recursive call is structurally
    smaller
dimSwap : (List t1, List t2) -> List (t1, t2)
dimSwap ([], _) = []
dimSwap (_, []) = []
dimSwap ((x::xs), (y::ys)) = (x,y)::(assert_total (dimSwap (xs, ys)))

generateGamesEF : (rounds : Nat) -> {auto ok : IsSucc rounds} -> (sigma : Signature) ->
    (s1, s2 : Structure sigma) ->
StructMorph (EFComonadObj rounds s1) s2 -> NEList (List (getType s1, getType s2))
generateGamesEF Z {ok = ItIsSucc} s s1 s2 morph impossible
```

```
generateGamesEF (S k) {ok = p} sig (t1 ** vs1 ** interp1 ** eq1) (t2 ** vs2 ** interp2
    ** eq2) morph =
   case (coextensionEF (S k) morph) of
      Smor mapping prof => (map (\pl => dimSwap (getList pl, getList (mapping pl)))
          (efplays (S k) vs1 ))

data Ex1Universe = One | Two | Three | Four

implementation [Ex1Equality] Eq Ex1Universe where
(==) One One = True
(==) Two Two = True
(==) Three Three = True
(==) Four Four = True
(==) _ _ = False

ex1EqCorrect : (a, b : Ex1Universe) -> True = (==) @{Ex1Equality} a b -> a = b
ex1EqCorrect One One Refl = Refl
ex1EqCorrect One Two Refl impossible
ex1EqCorrect One Three Refl impossible
ex1EqCorrect One Four Refl impossible
ex1EqCorrect Two One Refl impossible
ex1EqCorrect Two Two Refl = Refl
ex1EqCorrect Two Three Refl impossible
ex1EqCorrect Two Four Refl impossible
ex1EqCorrect Three One Refl impossible
ex1EqCorrect Three Two Refl impossible
ex1EqCorrect Three Three Refl = Refl
ex1EqCorrect Three Four Refl impossible
ex1EqCorrect Four One Refl impossible
ex1EqCorrect Four Two Refl impossible
ex1EqCorrect Four Three Refl impossible
ex1EqCorrect Four Four Refl = Refl

Ex1Next : Ex1Universe -> Ex1Universe
Ex1Next One = Two
Ex1Next Two = Three
Ex1Next Three = Four
Ex1Next Four = One

ex1NextChanges : (a : Ex1Universe) -> Not (a = Ex1Next a)
ex1NextChanges One Refl impossible
ex1NextChanges Two Refl impossible
ex1NextChanges Three Refl impossible
ex1NextChanges Four Refl impossible

ex1NextChangesTwice : (a : Ex1Universe) -> Not (a = Ex1Next (Ex1Next a))
ex1NextChangesTwice One Refl impossible
ex1NextChangesTwice Two Refl impossible
ex1NextChangesTwice Three Refl impossible
ex1NextChangesTwice Four Refl impossible

ex1NoMutualNext : (a, b : Ex1Universe) -> True = (==) @{Ex1Equality} (Ex1Next a) b ->
    True = (==) @{Ex1Equality} (Ex1Next b) a -> Void
ex1NoMutualNext a b prf1 prf2 =
```

```
    let p1 = ex1EqCorrect (Ex1Next a) b prf1 in let p2 = ex1EqCorrect (Ex1Next b) a
        prf2 in
    ex1NextChangesTwice a (sym (rewrite p1 in p2))

enumEx1Universe : NEList Ex1Universe
enumEx1Universe = listToNEL [One, Two, Three, Four]

Ex1Rel1 : Fin 1 -> Rel Ex1Universe
Ex1Rel1 FZ (One, Two) = True
Ex1Rel1 FZ (Two, Three) = True
Ex1Rel1 FZ (Three, Four) = True
Ex1Rel1 FZ _ = False

Ex1Rel2 : Fin 1 -> Rel Ex1Universe
Ex1Rel2 FZ (One, Two) = True
Ex1Rel2 FZ (Two, Three) = True
Ex1Rel2 FZ (Three, Four) = True
Ex1Rel2 FZ (Four, One) = True
Ex1Rel2 FZ _ = False

ex1rel1ImpliesNext : (a, b : Ex1Universe) -> True = Ex1Rel1 FZ (a, b) -> Ex1Next a = b
ex1rel1ImpliesNext One One Refl impossible
ex1rel1ImpliesNext One Two Refl = Refl
ex1rel1ImpliesNext One Three Refl impossible
ex1rel1ImpliesNext One Four Refl impossible
ex1rel1ImpliesNext Two One Refl impossible
ex1rel1ImpliesNext Two Two Refl impossible
ex1rel1ImpliesNext Two Three Refl = Refl
ex1rel1ImpliesNext Two Four Refl impossible
ex1rel1ImpliesNext Three One Refl impossible
ex1rel1ImpliesNext Three Two Refl impossible
ex1rel1ImpliesNext Three Three Refl impossible
ex1rel1ImpliesNext Three Four Refl = Refl
ex1rel1ImpliesNext Four One Refl impossible
ex1rel1ImpliesNext Four Two Refl impossible
ex1rel1ImpliesNext Four Three Refl impossible
ex1rel1ImpliesNext Four Four Refl impossible

ex1relTrueImpliesNotEqual : (a, b : Ex1Universe) -> True = Ex1Rel1 FZ (a, b) -> Not
    (True = (==) @{Ex1Equality} a b)
ex1relTrueImpliesNotEqual One One Refl prf2 impossible
ex1relTrueImpliesNotEqual One Two Refl Refl impossible
ex1relTrueImpliesNotEqual One Three Refl prf2 impossible
ex1relTrueImpliesNotEqual One Four Refl prf2 impossible
ex1relTrueImpliesNotEqual Two Two Refl prf2 impossible
ex1relTrueImpliesNotEqual Two One Refl prf2 impossible
ex1relTrueImpliesNotEqual Two Three Refl Refl impossible
ex1relTrueImpliesNotEqual Two Four Refl prf2 impossible
ex1relTrueImpliesNotEqual Three Three Refl prf2 impossible
ex1relTrueImpliesNotEqual Three One Refl prf2 impossible
ex1relTrueImpliesNotEqual Three Two Refl prf2 impossible
ex1relTrueImpliesNotEqual Three Four Refl Refl impossible
ex1relTrueImpliesNotEqual Four Four Refl prf2 impossible
ex1relTrueImpliesNotEqual Four One Refl prf2 impossible
```

```
ex1relTrueImpliesNotEqual Four Two Refl prf2 impossible
ex1relTrueImpliesNotEqual Four Three Refl prf2 impossible

ex2relTrueImpliesNotEqual : (a, b : Ex1Universe) -> True = Ex1Rel2 FZ (a, b) -> Not
    (True = (==) @{Ex1Equality} a b)
ex2relTrueImpliesNotEqual One One Refl prf2 impossible
ex2relTrueImpliesNotEqual One Two Refl Refl impossible
ex2relTrueImpliesNotEqual One Three Refl prf2 impossible
ex2relTrueImpliesNotEqual One Four Refl prf2 impossible
ex2relTrueImpliesNotEqual Two Two Refl prf2 impossible
ex2relTrueImpliesNotEqual Two One Refl prf2 impossible
ex2relTrueImpliesNotEqual Two Three Refl Refl impossible
ex2relTrueImpliesNotEqual Two Four Refl prf2 impossible
ex2relTrueImpliesNotEqual Three Three Refl prf2 impossible
ex2relTrueImpliesNotEqual Three One Refl prf2 impossible
ex2relTrueImpliesNotEqual Three Two Refl prf2 impossible
ex2relTrueImpliesNotEqual Three Four Refl Refl impossible
ex2relTrueImpliesNotEqual Four Four Refl prf2 impossible
ex2relTrueImpliesNotEqual Four One Refl Refl impossible
ex2relTrueImpliesNotEqual Four Two Refl prf2 impossible
ex2relTrueImpliesNotEqual Four Three Refl prf2 impossible

Ex1Structure1 : Structure 1
Ex1Structure1 = (Ex1Universe ** enumEx1Universe ** Ex1Rel1 ** Ex1Equality)

Ex1Structure2 : Structure 1
Ex1Structure2 = (Ex1Universe ** enumEx1Universe ** Ex1Rel2 ** Ex1Equality)

Ex1plays : NEList (EFplaysType 2 Ex1Universe)
Ex1plays = efplays 2 enumEx1Universe

Ex1Strategy : EFplaysType 2 Ex1Universe -> Ex1Universe
Ex1Strategy (MkPlays (FS FZ) [x]) = One
Ex1Strategy (MkPlays (FS (FS FZ)) [x, y]) with ((==) @{Ex1Equality} (Ex1Next x) y, (==)
    @{Ex1Equality} (Ex1Next y) x)
    | (False, False) = Three
    | (False, True) = Four
    | (True, False) = Two
    | (True, True) = One

equalityCongruence1 : Ex1Next xl = y -> y' = x -> a = (==) @{Ex1Equality} (Ex1Next xl)
    x -> a = (==) @{Ex1Equality} y y'
equalityCongruence1 prf1 prf2 prf3 = rewrite sym prf1 in (rewrite prf2 in prf3)

equalityReflexivity1 : (y : Ex1Universe) -> a = (==) @{Ex1Equality} y y -> a = True
equalityReflexivity1 One prf = prf
equalityReflexivity1 Two prf = prf
equalityReflexivity1 Three prf = prf
equalityReflexivity1 Four prf = prf

equalityCongruence2 : Ex1Next y = xl -> y = x -> a = (==) @{Ex1Equality} (Ex1Next x)
    xl' -> a = (==) @{Ex1Equality} xl xl'
equalityCongruence2 prf1 prf2 prf3 = rewrite sym prf1 in rewrite prf2 in prf3
```

```
stratProof : (a : EFplaysType 2 Ex1Universe) -> (b : EFplaysType 2 Ex1Universe) ->
(True = efRel @{Ex1Equality} (Ex1Rel1 FZ) (a, b)) -> True = Ex1Rel2 FZ (Ex1Strategy a,
    Ex1Strategy b)
stratProof (MkPlays (FS FZ) [x]) (MkPlays (FS FZ) [y]) prf =
   let contra = ex1relTrueImpliesNotEqual x y (conjunctsTrueL prf) in
   let diction = conjunctsTrueL (conjunctsTrueR {b = ((==) @{Ex1Equality} x y &&
       True)} prf) in void (contra diction)
stratProof (MkPlays (FS (FS FZ)) (x::[xl])) (MkPlays (FS FZ) [y]) prf with ((==)
    @{Ex1Equality} (Ex1Next x) xl, (==) @{Ex1Equality} (Ex1Next xl) x) proof p
   | (False, False) =
      let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (xl, y)} prf in
      let x1' = ex1rel1ImpliesNext xl y x1 in
      let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
          (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
      let x3 = pairsSplitR p in
      let test = equalityCongruence1 x1' x2 x3 in sym (equalityReflexivity1 y test)
   | (False, True) = Refl
   | (True, False) =
      let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (xl, y)} prf in
      let x1' = ex1rel1ImpliesNext xl y x1 in
      let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
          (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
      let x3 = pairsSplitR p in
      let test = equalityCongruence1 x1' x2 x3 in sym (equalityReflexivity1 y test)
   | (True, True) = void (ex1NoMutualNext x xl (pairsSplitL p) (pairsSplitR p))
stratProof (MkPlays (FS FZ) [y]) (MkPlays (FS (FS FZ)) (x::[xl])) prf with ((==)
    @{Ex1Equality} (Ex1Next x) xl, (==) @{Ex1Equality} (Ex1Next xl) x) proof p
   | (False, False) =
      let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (y, xl)} prf in
      let x1' = ex1rel1ImpliesNext y xl x1 in
      let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
          (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
      let x3 = pairsSplitL p in
      let test = equalityCongruence2 x1' x2 x3 in sym (equalityReflexivity1 xl test)
   | (False, True) =
      let x1 = conjunctsTrueL {a = Ex1Rel1 FZ (y, xl)} prf in
      let x1' = ex1rel1ImpliesNext y xl x1 in
      let x2 = ex1EqCorrect y x (conjunctsTrueL {a = (==) @{Ex1Equality} y x}
          (conjunctsTrueR {b = ((==) @{Ex1Equality} y x && True)} prf)) in
      let x3 = pairsSplitL p in
      let test = equalityCongruence2 x1' x2 x3 in sym (equalityReflexivity1 xl test)
   | (True, False) = Refl
   | (True, True) = void (ex1NoMutualNext x xl (pairsSplitL p) (pairsSplitR p))
stratProof (MkPlays (FS (FS FZ)) (x::[xl])) (MkPlays (FS (FS FZ)) (y::[yl])) prf =
   let x1 = ex1rel1ImpliesNext xl yl (conjunctsTrueL prf {a = Ex1Rel1 FZ (xl, yl)}) in
   let x2 = ex1EqCorrect xl yl (conjunctsTrueL (conjunctsTrueR (conjunctsTrueR prf)))
       in
   void (ex1NextChanges xl (sym (trans x1 (sym x2))))

Ex1StrategyMorph : StructMorph (EFComonadObj 2 Ex1Structure1) Ex1Structure2
Ex1StrategyMorph = Smor Ex1Strategy (ItIsStructMorph prf)
where prf : (k : Fin 1) -> IsGraphMorph Ex1Strategy (efInterp @{Ex1Equality} 2 Ex1Rel1
    k) (Ex1Rel2 k)
prf FZ = IsGraphMorphByElem stratProof
```

```
Ex1Games : NEList (List (Ex1Universe, Ex1Universe))
Ex1Games = generateGamesEF 2 1 Ex1Structure1 Ex1Structure2 Ex1StrategyMorph
```

Bibliography

[1] Samson Abramsky, Anuj Dawar, and Pengming Wang. The pebbling comonad in finite model theory. *CoRR*, abs/1704.05124, 2017.

[2] Samson Abramsky and Nihil Shah. Relating structure and power: Comonadic semantics for computational resources. 2018.

[3] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[4] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[5] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications Company, 2017.

[6] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[7] Per Martin-Lof. Intuitionistic type theory, 1980. Notes by Giovanni Sambin of a series of lectures given in Padua.

[8] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[9] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[10] B.C. Pierce and Carnegie-Mellon University. School of Computer Science. *A Taste of Category Theory for Computer Scientists*. CMU-CS. School of Computer Science, Carnegie Mellon University, 1990.

[11] Keith Pinson. Non-empty-list comonad. https://stackoverflow.com/questions/46554099/non-empty-list-comonad.